Python Parallel Programming Cookbook

Python并行编程手册

[意] Giancarlo Zaccone 著
张龙 宋秉金 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

若想充分利用所有的计算资源来构建高效的软件系统，并行编程技术是不可或缺的一项技能。本书以Python为蓝本，对并行编程领域的各项技术与知识进行了广泛且深入的讲解。通过对本书的学习，读者将能够快速且准确地掌握并行编程方方面面的技能，从而应用在自己的项目开发中，提升系统运行效率。

本书共分为6章，从原理到实践系统化地对并行编程技术进行了层层剖析，并通过大量可运行的实例演示了每一个知识点的具体运用方式，是提升并行编程技能的一本不可多得的好书。相信本书的出版将会填补Python在并行编程领域应用的一大空白，能够帮助想要从事并行编程与并行计算的读者提升实践能力，并将这一能力应用到实际的项目开发中。

版权贸易合同登记号 图字：01-2015-8415

图书在版编目（CIP）数据

Python 并行编程手册 /（意）詹卡洛 · 扎克尼（Giancarlo Zaccone）著；张龙，宋秉金译 . —北京：电子工业出版社，2018.4
书名原文：Python Parallel Programming Cookbook
ISBN 978-7-121-33753-6

Ⅰ . ① P… Ⅱ . ①詹… ②张… ③宋… Ⅲ . ①软件工具—程序设计—手册 Ⅳ . ① TP311.561-62

中国版本图书馆 CIP 数据核字（2018）第 036159 号

策划编辑：许　艳
责任编辑：刘　舫
印　　刷：三河市君旺印务有限公司
装　　订：三河市君旺印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787 × 980　1/16　　印张：15.25　　字数：361千字
版　　次：2018年4月第1版
印　　次：2018年4月第1次印刷
定　　价：59.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

译者序

今日之时代是并行编程与多核的时代，硬件成本越来越低，如何充分利用硬件所提供的各种资源是每一个软件开发者需要深入思考的问题。在并行编程的浪潮下，每一个有追求的软件开发者都应该拥抱这种变化，转换编程思维，使程序能够在多核及并行的环境下高效运行。

本书是一本专门介绍并行编程的图书。通过对硬件与软件的剖析，总结出了影响并行编程的多种因素，并通过大量的理论与实践来分析如何实现并行编程，以及如何更好地利用硬件资源来提升程序并行运行的效率。

值得一提的是，本书以 Python 语言作为范本对并行编程进行了深入的讲解。Python 是一门简洁且优雅的语言，目前的发展势头非常迅猛，在很多领域都能见到它的身影。以 Python 语言作为示范语言来讲解并行编程是非常恰当的。

本书首先对 Python 编程语言与并行编程理论进行了较为详尽的讲解，然后介绍了基于线程与基于进程的并行编程方式。通过对这两种方式进行介绍，读者可以迅速进入并行编程领域，掌握并行编程的一般方法与技巧，同时还会对 Python 中涉及线程与进程的模块有所了解，在后半部分，本书对异步编程与分布式 Python 以及如何使用 Python 进行 GPU 编程进行了深入的讲解，这些内容将会帮助读者实现从了解并行编程到深入掌握并行编程的蜕变。

值得一提的是，全书在理论讲解过程中辅以大量可运行的代码示例。这些示例非常好地演示了所讲理论的实际运用，读者可以通过这些示例加深对并行编程相关理论的理解与认识，也可以对这些示例稍加修改应用到实际的项目开发过程当中。

并行编程是一个较为复杂的技术领域，常常令很多开发者望而却步。不过，本书的出现将会改变这一局面。通过对本书的学习，读者将会快速且准确地建立起对并行编程的认知，并进

一步加强对并行编程的理解；此外，读者还可以通过阅读本书，掌握用 Python 实现并行编程的各种方式与技巧，并形成良好的并行编程思维方式。

全书由张龙与宋秉金翻译完成，其中张龙翻译了第 1~3 章的内容，宋秉金翻译了第 4~6 章的内容。在图书翻译过程中，得到了电子工业出版社计算机出版分社的许艳、张春雨二位老师的大力协助，在此向二位老师表示深深的感谢。二位老师在专业素养与团队协调方面展现出了极高的专业性，确保了本书的翻译工作能够顺利完成。每次与二位老师沟通都非常顺畅，同时进一步确保了译稿的质量。

虽已尽心尽力，奈何技术与文字水平有限；虽已校对多次，但依然不敢保证全书没有任何错误。因此，读者在阅读本书的过程中如果发现任何问题都请不吝赐教，可以通过 zhanglong217@163.com 联系我，以期再版时改进。

最后，衷心期望本书能给希望系统学习并行编程的读者带来切实的帮助，帮助大家准确建立起并行编程的思维方式，并能应用到实际的项目开发当中。

张龙

2018 年 2 月 8 日于北京

关于作者

Giancarlo Zaccone 拥有超过 10 年的管理研发项目的经验，涉及科学与工业这两个领域。他曾以研究员身份就职于国家研究委员会（CNR），主要从事一些并行科学计算与科学可视化项目。

他目前作为一名软件工程师就职于一家咨询公司，主要负责开发和维护一些面向太空和防御应用的软件系统。

Giancarlo 拥有那不勒斯费德里科二世大学的物理学硕士学位，并且获得了罗马大学科学计算专业的第二研究生学位。

可以通过 https://it.linkedin.com/in/giancarlozaccone 了解到关于 Giancarlo 的更多信息。

关于审校者

Aditya Avinash 是一名研究生，专注在计算机图形学与GPU上。他感兴趣的领域有编译器、驱动程序、基于物理学的渲染以及实时渲染等。他目前正在为 MESA（Linux 的开源图形驱动栈）贡献力量，主要负责实现 AMD 后端的 OpenGL 扩展。这是他真正感兴趣的地方。他还喜欢编写编译器，将高层次抽象代码转换为 GPU 代码。他开发了 Urutu，它可以使用 Python 赋予 GPU 线程级的并行能力。出于这一点，NVIDIA 资助了他一对 Tesla K40 GPU。目前，他致力于 RockChuck 的开发工作，负责根据不同后端将 Python 代码（使用数据并行抽象编写）转换为 GPU/CPU 代码。这是他听取了大量 Python 开发者希望实现针对 Python 与 GPU 的数据并行抽象的建议而启动的一个项目。

Aditya 拥有计算机工程背景，设计了硬件与软件来适应某些应用（ASIC）。基于这一点，他获得了关于如何使用 FPGA 与 HDL 的经验。此外，他主要使用 Python 与 C++ 编写代码。在 C++ 中，他使用过 OpenGL、CUDA、OpenCL 以及其他多核编程 API。由于他还是一名学生，因此他的大多数工作并不隶属于任何机构或个人。

Ravi Chityala 是 Elekta Inc 的一名高级工程师。他在图像处理与科学计算领域拥有超过 12 年的经验。他还是加利福尼亚大学的兼职讲师，负责讲授 Python 高级编程。一开始他将 Python 当作脚本工具来使用，并且喜欢上了这门简单、强大、富有表述力的语言。他现在使用 Python 进行 Web 开发、科学原型制造与计算，并将其作为胶水来自动化这个过程。他将自己在图像处理上的经验与对 Python 的热爱结合起来，与他人合著了图书 *Image Acquisition and Processing using Python*，该书由 CRC 出版社出版。

Mike Galloy 是一名软件开发者，他专注于高性能计算与科学编程中的可视化。他在工作中主要使用 IDL，不过偶尔也会使用 C、CUDA 与 Python。他目前在位于茂纳罗亚太阳天文台的国家大气研究中心（NCAR）工作。在这之前，他供职于 Tech-X 公司，是 GPULib 的主力开发者，GPULib 是一个针对 GPU 加速计算的 IDL 绑定库。他是开源项目 IDLdoc、mgunit 与 rIDL 的创建者与主力开发者，同时也是图书 *Modern IDL* 的作者。

Ludovic Gasc 是 Eyepea and ALLOcloud 公司的高级软件开发者与工程师，该公司是欧洲知名的开源 VoIP 与统一通信公司。

在过去的 5 年间，他基于 Python、AsyncIO、PostgreSQL 与 Redis 为电信部门开发了多款分布式系统。

可以通过他的博客 `http://www.gmludo.eu` 联系到他。

他还是博文 *API-Hour: Write efficient network daemons (HTTP, SSH) with ease* 的作者。要想了解更多信息，请访问 `http://www.api-hour.io`。

前言

计算机科学的研究不仅应该涵盖计算处理所基于的原则，还应该反映出这些领域当前的知识状态。时至今日，技术的发展要求来自计算机科学所有分支领域的专家通晓软件与硬件，它们之间的交互是理解计算处理基本原理的关键所在。

出于这个原因，本书特别关注硬件架构与软件之间的关系。

之前，程序员可以借助于硬件设计、编译器与芯片让其软件程序在不做任何修改的情况下，速度变得更快或效率更高。

这个时代已然结束。现在，如果希望程序运行速度能变得更快，那么它必须要成为一个并行程序。

虽然很多研究人员的目标是程序员不需要了解运行其程序的硬件所具有的并行性，不过要想实现这个梦想还需要很多年的时间。今天，大多数程序员还是需要透彻理解硬件与软件之间的联系，这样程序才能在现代计算机架构上高效运行。

为了介绍并行编程的概念，我们使用了 Python 编程语言。Python 很有趣，且易于使用，其流行度在近几年稳步上升。Python 是由 Guido van Rossum 在 10 多年前开发出来的，Python 语法的简洁性与易于使用的特性很大程度上来源于 ABC，这是一门于 20 世纪 80 年代开发出来的教学语言。

除了这个具体的上下文外，Python 还被用于解决实际的问题，它从一些典型的编程语言中借鉴了很多特性，比如说 C ++、Java 与 Scheme 等。这是其最卓越的特性之一，使得它广受专业软件开发者、科学研究产业以及计算机科学教育者的青睐。之所以有这么多人喜欢 Python，一个原因是它在实践与概念之间提供了最佳的平衡。Python 是一门解释型语言，因此无须陷入

编译与链接的泥潭中就可以立刻动手做事情。Python 还提供了广泛的软件库，可用在从 Web 开发、图形学到并行计算的各种任务中。这种侧重于实践的特性非常吸引广大读者，可以让大家运行本书所介绍的重要项目。

本书提供了大量的示例，这些示例的灵感来源于很多场景，可以用它们解决实际问题。本书介绍了并行架构的软件设计原则，并确保程序清晰可读，避免使用一些复杂技术，都是一些简单且直接的示例。每个主题都以一个完整、可运行的 Python 程序的一个片段来阐述，后面跟着的是程序的输出结果。

书中各章节采用模块化的方式组织，可以从最简单的讨论跳到高级的主题，这么做也适合于那些只想学习一些特定主题的读者。

我希望本书这样的结构与内容安排能够帮助读者更好地理解和掌握并行编程技术。

本书主要内容

第 1 章概览了并行编程架构与编程模型。本章介绍了 Python 编程语言、语言的特性、其易于使用和学习的特点、可扩展性以及丰富的软件库与应用。此外，本章还介绍了如何在应用以及并行计算中用好 Python 这个工具。

第 2 章介绍了如何通过 Python 线程模块来实现线程并行。通过完整的编程示例，读者将学习到如何同步和操纵线程来实现多线程应用。

第 3 章介绍了基于进程的用于并行化程序的方式。本章通过完整的示例展示了如何使用 Python 多线程模块。此外，本章还介绍了如何通过进程来实现通信，借助 Python 的 mpi4py 模块使用消息传递的并行编程范式。

第 4 章介绍了并发编程的异步模式。在某些方面，它要比基于线程的方式简单一些，因为它提供了单指令流，任务会明确放弃控制而非武断地挂起。本章介绍了如何通过 Python asyncio 模块将每个任务组织为一系列更小的步骤，并在异步模式下执行。

第 5 章介绍了分布式计算。它指的是从逻辑上（甚至可能是地理位置上分布的）聚合几个计算单元，并以一种透明且一致的方式协同运行单个计算任务的过程。本章将会介绍 Python 所提供的，使用面向对象、Celery、SCOOP、远程过程调用的架构实现解决方案，比如，Pyro4 与 RPyC。本章也介绍了其他不同的方式，如 PyCSP、Disco，后者是 Python 版本的 MapReduce 算法。

第 6 章介绍了现代图形处理单元（GPU），它使数值计算的性能有了突破性提升，代价则是编程复杂度的增加。实际上，GPU 编程模型要求程序员手工管理 CPU 与 GPU 之间的数据传输。本章将会通过程序示例与用例介绍如何使用 GPU 卡所提供的计算能力，其中使用了强大的 Python 模块：PyCUDA、NumbaPro 与 PyOpenlCL。

阅读前的准备

本书所有示例都已经在 Windows 7 的 32 位机器上进行过测试。此外，使用 Linux 环境将会对学习大有裨益。

运行示例所需的 Python 版本是：

- Python 3.3（前 5 章）
- Python 2.7（只有第 6 章需要）

需要使用到如下模块（都是可以自由下载的）：

- mpich-3.1.4
- pip 6.1.1
- mpi4py1.3.1
- asyncio 3.4.3
- Celery 3.1.18
- Numpy 1.9.2
- Flower 0.8.32（可选）
- SCOOP 0.7.2
- Pyro 4.4.36
- PyCSP 0.9.0
- DISCO 0.5.2
- RPyC 3.3.0
- PyCUDA 2015.1.2
- CUDA Toolkit 4.2.9（最低为此版本）
- NVIDIA GPU SDK 4.2.9（最低为此版本）
- NVIDIA GPU 驱动

- Microsoft Visual Studio 2008 C++ Express Edition（最低为此版本）
- Anaconda Python Distribution
- NumbaPro 编译器
- PyOpenCL 2015.1
- Win32 OpenCL Driver 15.1（最低为此版本）

本书面向的读者

本书是写给那些想要使用并行编程技术来编写强大且高效代码的开发者的。阅读完本书后，读者将掌握并行计算的基础知识与高级特性。Python 编程语言易于学习，即便不是专家也能够轻松理解本书所介绍的主题。

本书结构

本书包含如下组成部分。

准备工作

这部分介绍了攻略的主题，描述了如何为该攻略搭建好软件或是进行一些设置。

具体操作

这部分介绍了实现攻略所需要的步骤。

实例精解

这部分通常会对“具体操作”部分的内容进行解释和说明。

知识扩展

这部分包含了关于攻略的一些额外信息，目的在于使读者更加深入地理解攻略的内容。

参考

这部分包含一些关于攻略的参考信息。

约定

在本书中，你会看到各种样式的文本，用于区分不同类型的信息。下面对这些文本样式及其含义进行一些说明。

文本中的代码、数据库表名、目录名、文件名、文件扩展、路径名、虚拟的URL、用户输入以及Twitter handle会这样表示“要执行第一个示例，我们需要用到程序`helloPython-WithThreads.py`.”。

代码块样式如下所示：

```
print ("Hello Python Parallel Cookbook!!")
closeInput = raw_input("Press ENTER to exit")
print "Closing calledProcess"
```

希望读者注意到代码块中的某一部分时，相关行或是代码会加粗处理：

```
@asyncio.coroutine
def factorial(number):
do Something

@asyncio.coroutine
```

任何命令行输入或是输出的样式如下所示：

```
C:\>mpiexec -n 4 python virtualTopology.py
```

警告或是重要的说明位于框中。

提示与技巧位于这个图标后。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在下载资源处下载。
- **提交勘误**：你对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方读者评论处留下你的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33753

目录

1
并行计算与Python起步

本章主要内容：

- 何为并行计算
- 并行计算内存架构
- 内存组织
- 并行编程模型
- 如何设计并行程序
- 如何评估并行程序的性能
- Python 简介
- 并行世界中的 Python
- 进程与线程介绍
- 开始使用进程与 Python
- 开始使用线程与 Python

介绍

本章将会概述并行编程架构与编程模型。这些概念对于初次接触并行编程技术、经验不太丰富的程序员来说非常有价值。对于有经验的程序员来说，本章内容可以作为一个基本的参考。本章还会介绍并行系统的双重特性。第一个特性基于系统架构，第二个特性基于并行编程范式。对于程序员来说，并行编程总是充满挑战的。这种基于编程的方式还会在本章后面介绍并行程序的设计过程时进一步阐述。本章最后将会对 Python 编程语言进行简短的介绍。这门语言的诸多特性（比如易于使用和学习，可扩展性以及丰富的软件库与应用）使得它对于任何应用来说都是一个颇有价值的工具，当然，对于并行计算亦如此。本章的最后一部分将会介绍线程与进程的概念，以及它们在 Python 语言中的使用。解决大问题的一种典型方式是将其分解为一系列小问题以及独立的部分，这样就可以同时解决它们了。并行程序就是针对使用这种方式的

程序而设计的，也就是说，对于一个一般性任务来说，使用多个处理器同时工作。每个处理器都处理属于它自己的那部分问题（独立的部分）。此外，处理器之间的数据信息交换可以发生在计算过程中。时至今日，很多软件应用都需要更多的计算能力。达成该目标的一种方式就是提高处理器的时钟频率或是增加每个芯片上的处理器核心数。提高时钟频率会增加散热，从而降低每瓦特的性能；此外，这还需要特殊的冷却设备。增加核心数只不过是一个看起来可行的方案，因为能量功耗与消耗还远远没有达到极限，在性能上并没有那么明显的提升。

为了解决这个问题，计算机硬件厂商决定采用多核架构，即单个芯片上包含两个或多个处理器（核心）。另一方面，GPU 制造商还引入了基于多计算核心的硬件架构。实际上，现如今的计算机几乎都是采用多个以及异构的计算单元，每个单元都由不同数量的核心构成，比如，最常见的多核架构。

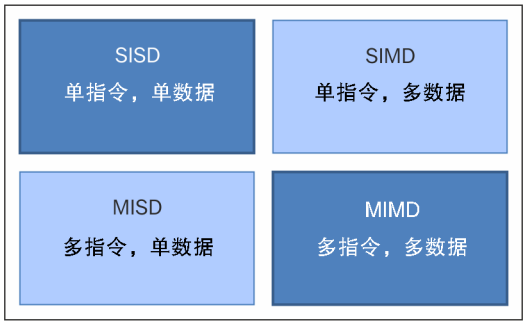
因此，我们非常有必要采用并行计算的编程范式、技术以及工具使用可用的计算资源来。

并行计算内存架构

根据可同时处理的指令数量与数据量的不同，计算机系统可以划分为如下 4 类：

- 单指令，单数据（SISD）
- 单指令，多数据（SIMD）
- 多指令，单数据（MISD）
- 多指令，多数据（MIMD）

这种分类方式叫作费林分类（Flynn’s taxonomy），如下图所示。



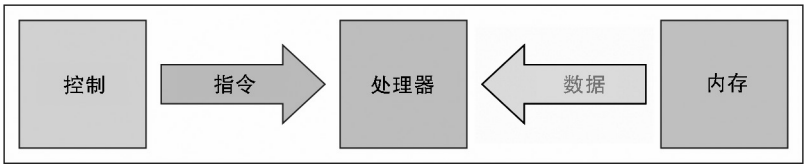
SISD

SISD 计算系统是一个单处理器机器，它所执行的每个指令都只会操作单个数据流。在 SISD 中，机器指令是顺序处理的。

在一个时钟周期中，CPU 会执行如下操作。

- **获取**：CPU 从内存区域获取数据与指令，该内存区域叫作寄存器。
- **解码**：CPU 对指令进行解码。
- **执行**：指令在数据上得到执行。操作结果会被存储到另一个寄存器中。

当执行阶段完成后，CPU 会对自身进行设置，从而开始另一个 CPU 周期，如下图所示。



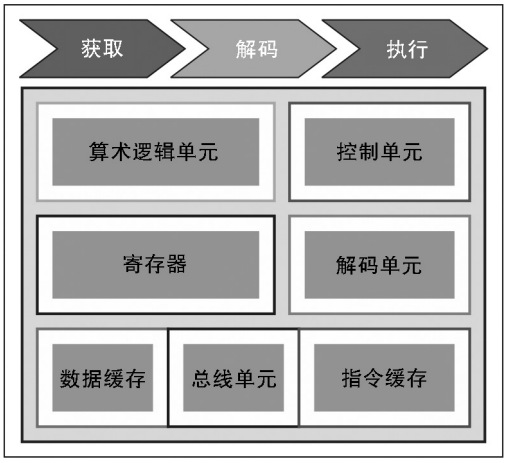
SISD 架构模式

在这类计算机中执行的算法是顺序的（又叫作串行），因为它们并不包含任何并行。只拥有单个 CPU 的硬件系统就是 SISD 计算机。

构成这种架构（冯 · 诺伊曼架构）的主要元素有如下几项。

- **中央存储单元**：用于存储指令与程序数据。
- **CPU**：用于从存储单元中获取指令与（或）数据，它会对指令进行解码并顺序地执行它们。
- **I/O 系统**：指的是程序的输入与输出数据。

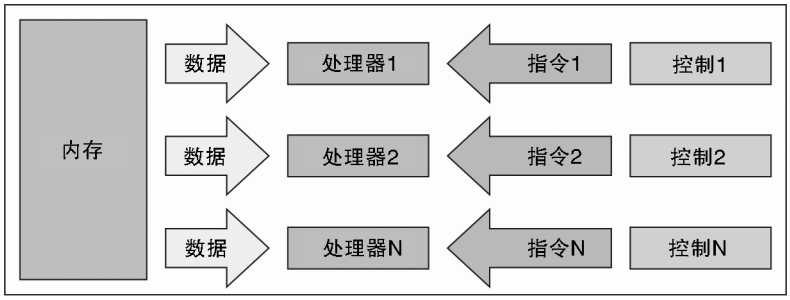
传统的单处理器计算机被归类为 SISD 系统。下图特别展示了在获取、解码与执行阶段中分别用到了 CPU 的哪些区域。



在获取 - 解码 - 执行阶段所用到的 CPU 组件

MISD

该模型中的 *n* 个处理器，每一个都有自己的控制单元，它们共享同一个存储单元，如下图所示。在每个时钟周期内，从内存接收到的数据会被所有处理器同时处理，每个处理器都会按照从其控制单元中所接收到的指令顺序进行处理。在这种情况下，通过对相同的数据执行几个操作实现了并行（指令级并行）。这种架构所能有效解决的问题是相当特殊的，比如说与数据加密相关的问题等；出于这个原因，计算机 MISD 并未在商业上流行起来。MISD 计算机更多的是用在智力训练而非实际使用。



MISD 架构模式

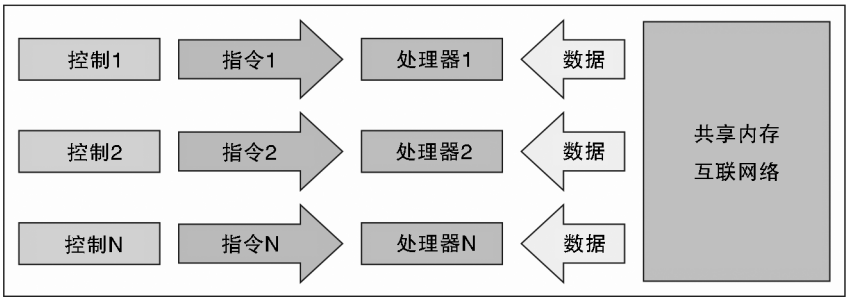
SIMD

SIMD 计算机包含 *n* 个相同的处理器，每个处理器都有自己的本地内存，可以用于存储数据。所有处理器都处于单个指令流的控制之下；此外，还有 *n* 个数据流，分别针对每个处理器。在每个步骤中，处理器都会同时工作并执行相同的指令，不过是在不同的数据元素上执行。这是一种数据级的并行。SIMD 架构要比 MISD 架构更加通用。大量应用所涉及的诸多问题都可以通过 SIMD 计算机中的并行算法来解决。另一个有趣的特性是这些计算机所用算法的设计、分析与实现相对来说都比较容易。局限在于只有那些可以被分解为一系列子问题的问题（这些子问题要完全一样，每个子问题后面会通过相同的指令集同时解决）才能通过 SIMD 计算机来解决。对于根据该范式所开发出的超级计算机来说，我们必须要提到的就是 Connection Machine（1985 Thinking Machine）与 MPP（NASA—1983）。第 6 章将会提到，拥有大量 SIMD 嵌入式单元的现代图形处理器单元（GPU）的出现使得这种计算范式得到了更为广泛的使用。

MIMD

根据费林的分类，这种并行计算机是最为通用，也是更为强大的一种，如下图所示。它有 *n* 个处理器、*n* 个指令流以及 *n* 个数据流。每个处理器都有自己的控制单元与本地内存，从计算角度来说，这使得 MIMD 架构要比 SIMD 更加强大。每个处理器都是在它自己的控制单元

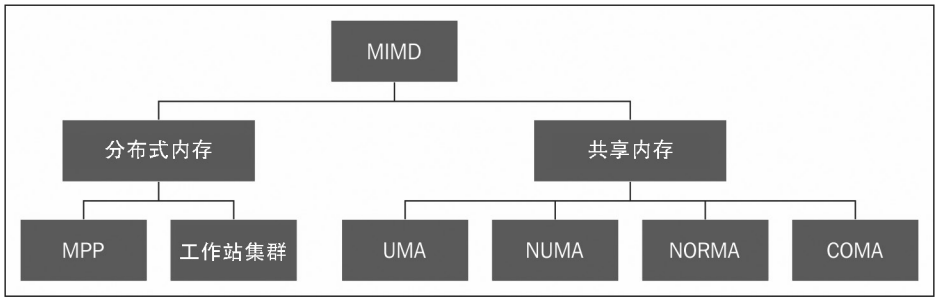
所发出的指令流的控制下来进行操作的；因此，处理器可以对不同的数据执行不同的程序，可以将一个大问题分解为多个不同的子问题，然后加以解决。MIMD 架构是通过线程与（或）进程级别的并行的帮助来实现的。这还意味着，处理器通常会异步执行。这类计算机用于解决那些拥有不规则结构的问题，而 SIMD 则要求问题的结构要规则才行。时至今日，这种架构已经应用到了很多 PC、超级计算机以及计算机网络中。不过，有一点需要注意，异步算法的设计、分析与实现不是那么容易的事情。



MIMD 架构模式

内存组织

在评估并行架构时需要考虑的另一个方面就是内存组织了，更准确地说是数据的访问方式。无论处理单元速度多么快，如果内存无法以足够的速度来维护并提供指令和数据，那么在性能上也不会有什么改进。要想让内存的响应时间跟得上处理器的速度，我们需要解决的主要问题就是存储周期时间，它指的是连续两个操作之间所经过的时间。处理器的周期时间通常要比内存的周期时间短。当处理器开始传输数据时（向内存传输或是从内存获取），内存将会在整个存储周期内被占用：在这期间，其他设备（I/O 控制器、处理器，甚至是发出该请求的处理器自身）都无法使用内存，因为它要对请求做出响应。下图所示为 MIMD 架构中的内存组织。

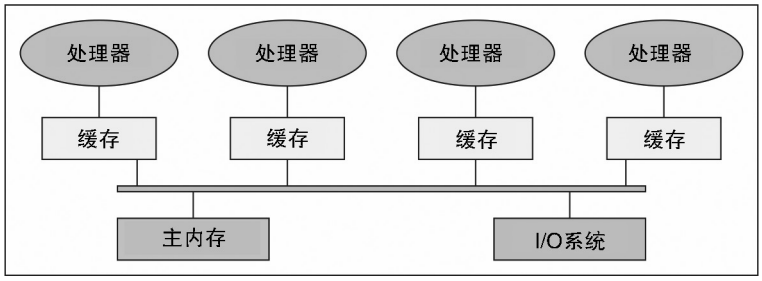


MIMD 架构中的内存组织

对内存访问问题的解决方案导致了MIMD架构的分歧。在第一类系统中(即共享内存系统)有一个高层的虚拟内存，所有处理器都可以访问该内存中的数据与指令。另一类系统则使用了分布式内存模型，其中每个处理器都有自己的本地内存，其他处理器是无法访问的。共享内存与分布式内存之间的差别在于虚拟内存或是从处理器的视角所看到的内存结构。从物理上来说，几乎每个系统内存都会被划分为不同的组件，它们之间的访问是独立的。共享内存与分布式内存的区别则在于处理器单元的内存访问管理方式。如果处理器要执行指令 `load R0, i`，这表示将内存位置 `i` 的内容加载到 `R0` 寄存器中，问题在于会发生什么呢？在共享内存系统中，`i` 索引是一个全局地址，内存位置 `i` 对于每个处理器来说都是一样的。如果两个处理器同时执行该指令，那么它们就会在寄存器 `R0` 中加载相同的信息。在分布式内存系统中，`i` 是一个本地地址。如果两个处理器同时加载语句 `R0`，那么在各自的寄存器 `R0` 中就会有不同的值，因为在这种情况下，每个本地地址会被分配不同的内存单元。共享内存与分布式内存之间的差别对于开发者来说至关重要，因为它决定了并行程序的不同部分之间的通信方式。在一个系统中，共享内存足以在内存中构建数据结构，然后进入并行子程序，它们是该数据结构的引用变量。此外，分布式内存机器必须要在各自的本地内存中复制共享数据。这些副本是通过从一个处理器向另一个处理器发送包含数据的消息来创建的。这种内存组织的一个缺点在于，有时这些消息会非常大，需要花费相对较长的时间来传递。

共享内存

下图展示了共享内存多处理器系统的模式。这里的物理连接是相当简单的。总线结构可以支持共享相同通道的任意数量的设备。总线协议最初被设计为支持单个处理器，一个或多个磁盘以及磁带处理器可以通过这里的共享内存进行通信。注意，每个处理器都会有一个与之关联的高速缓存，因为很多时候处理器都需要本地内存中的数据或是指令。当一个处理器修改了同时被其他处理器所用的内存系统中的数据时就会产生问题。新值需要从处理器缓存中传递过来，而处理器缓存中的值已经被修改到了共享内存中；不过，接下来它还需要传递给所有其他的处理器，这样老的值就无法正常使用了。这个问题叫作缓存一致性问题，这是内存一致性的一个特例，需要硬件实现来处理并发问题与同步，类似于线程编程一样。



共享内存架构模式

共享内存系统的主要特点如下所示。

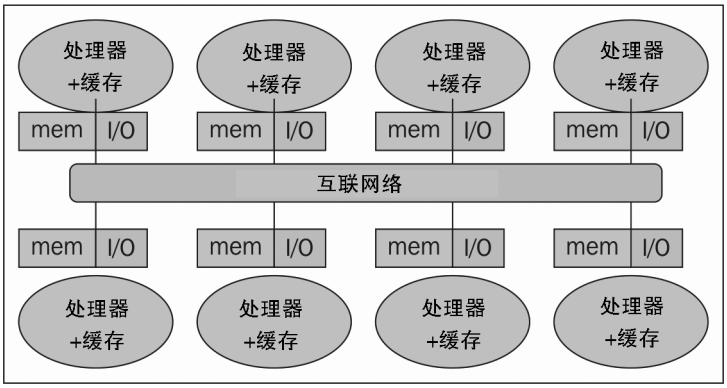
- 内存对于所有处理器来说都是一样的，比如，与相同数据结构所关联的所有处理器都会使用同样的逻辑内存地址，这样就会访问到相同的内存位置。
- 同步是通过控制处理器对共享内存的访问来实现的。实际上，在某一时刻只有一个处理器能够访问到内存资源。
- 当一个任务在访问共享内存时，另一个任务是不可以修改共享内存的位置的。
- 数据共享是非常快的；两个任务间通信所需的时间等于读取单个内存位置的时间（取决于内存访问的速度）。

共享内存系统中的内存访问如下所示。

- **统一内存访问（UMA）**：该系统的基本特点是，对于每个处理器以及内存的任何区域来说，对内存的访问时间都是恒定的。出于这个原因，这些系统又叫作*对称多处理器*（SMP）。这些系统相对来说比较容易实现，不过可伸缩性并不好；程序员需要负责同步的管理，这是通过在管理资源的程序中插入恰当的控制、信号量以及锁来实现的。
- **非统一内存访问（NUMA）**：该架构将内存区域划分为高速访问区域（分配给每个处理器，是数据交换的公共区域）以及低速访问区域两种。这些系统又叫作*分布式共享内存系统*（DSM）。其可伸缩性非常好，不过开发起来难度颇大。
- **无远程内存访问（NORMA）**：内存在物理上被分配给各个处理器（本地内存）。所有的本地内存都是私有的，只能为本地处理器所访问。处理器之间的通信是通过用于消息交换的通信协议来实现的，叫作消息传递协议。
- **仅缓存访问（COMA）**：这些系统只有缓存。在分析 NUMA 架构时，我们知道，其架构是将数据的副本保存到缓存中，并且这些数据在主内存中还会有一个副本存在。该架构去除了主内存中的副本，只保留缓存，内存在物理上被分配给了各个处理器（即本地内存）。所有的本地内存都是私有的，只能为本地处理器所访问。处理器之间的通信是通过用于消息交换的通信协议来实现的，叫作消息传递协议。

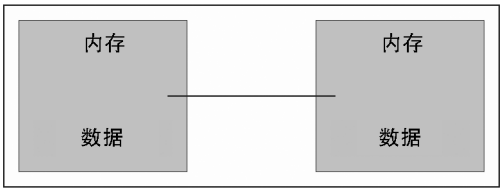
分布式内存

在分布式内存系统中，内存与每个处理器关联到了一起，一个处理器只能访问到它自己的内存。一些作者将这类系统叫作“多计算机系统”，这反映了这样一个事实，即系统元素本身是小型且完备的处理器与内存系统，如下图所示。



分布式内存架构模式

这种组织方式有几个优点。首先，在通信总线或是开关层面上不会再出现冲突。每个处理器都可以使用其自己本地内存的全部带宽而不会妨碍其他处理器。其次，无公共总线意味着对于处理器的数量不会再有固有的限制，系统的大小只受限于连接处理器的网络。第三，不存在缓存一致性的问题了。每个处理器负责管理自己的数据，不必再考虑副本的更新问题。主要的缺点则是处理器之间的通信更加难以实现。如果一个处理器需要另一个处理器内存中的数据，那么这两个处理器就需要通过消息传递协议来交换消息。这会引入两个速度降低之源；一个处理器构建消息并向另一个处理器发送消息需要时间；另外，处理器还得停下来管理从其他处理器所接收到的消息。针对分布式内存机器所设计的程序需要组织为一组独立的任务，这些任务间通过消息进行通信，如下图所示。



基本的消息传递

分布式内存系统的主要特性如下所示。

- 物理上，内存在处理器之间是分布式的；每个本地内存都只会被其处理器所直接访问。
- 同步是通过在处理器间（通信）移动数据（即便只是消息本身亦如此）来实现的。
- 本地内存中数据的分割会影响机器的性能——划分的精确性非常重要，因为这样会将CPU 之间的通信降到最低。除此之外，用于协调这些分解与组合操作的处理器必须能与对数据结构的每一部分进行操作的处理器高效通信。

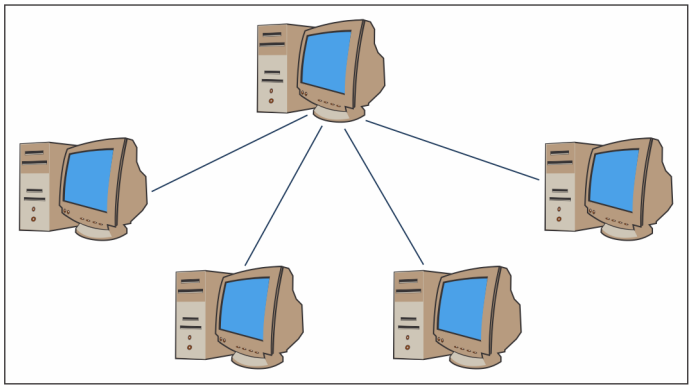
- 使用消息传递协议，这样 CPU 就可以通过数据包的交换来彼此通信。消息是信息的离散单元；它们拥有定义明确的身份，这样就可以对其进行区分了。

大规模并行处理

MPP 机器由成百上千个处理器组成（在一些机器上的规模可以达到成千上万），这些处理器之间通过通信网络进行连接。世界上最快的计算机就是基于这样的架构的，比如，Earth Simulator、Blue Gene、ASCI White、ASCI Red、ASCI Purple 及 Red Storm 等。

工作站集群

这些处理系统基于传统计算机，它们之间通过通信网络进行连接。计算集群就属于这类，如下图所示。



工作站集群架构示例

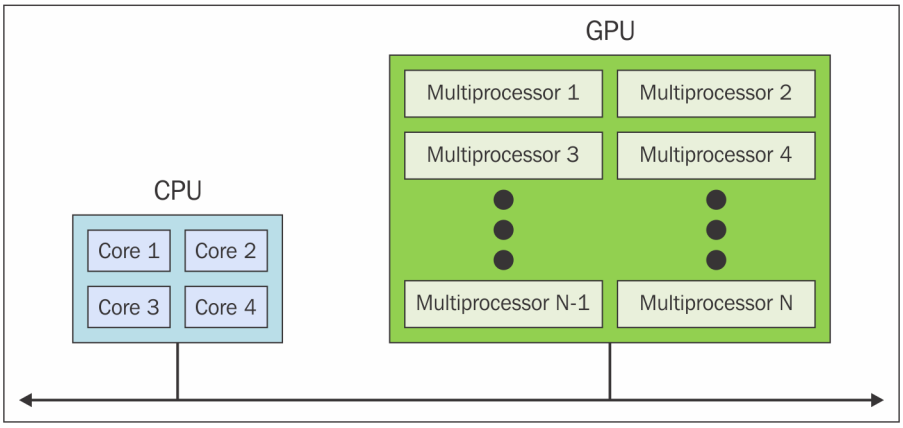
在集群架构中，我们将节点定义为集群中的单个计算单元。对于用户来说，集群是完全透明的——所有的硬件与软件的复杂性都被隐藏起来，我们在访问数据与应用时就好像它们都来自于单个节点一样。

下面介绍三类集群。

- **容错集群**：在该类集群中，节点的活动会被持续监控。当一个节点停止工作时，另一台机器会接管它的活动。其目标旨在通过架构的冗余来确保持续的服务。
- **负载均衡集群**：在该系统中，工作请求会被发送给活动较少的节点。这确保了完成整个过程所需的时间会更少一些。
- **高性能计算集群**：在该类集群中，每个节点都会被配置以提供非常高的性能。整个过程依然会被划分为在多个节点上执行的多个任务。任务是并行化的，并且分布在不同的机器上。

异构架构

超级计算机同构世界中引入的 GPU 加速器改变了之前超级计算机使用与编程方式的本质。尽管 GPU 提供了很高的性能，不过它们无法作为一种自治的处理单元，因为它们总是要与 CPU 协同使用才行。因此，编程范式非常简单；CPU 以串行方式进行控制与计算，将计算代价高昂并且需要很高并行度的任务分配给图形加速器。CPU 与 GPU 之间的通信不仅可以通过高速总线来实现，还可以通过针对物理或是虚拟的单个内存区域的共享来实现。实际上，如果两台设备都没有自己的内存区域，那么可以通过各种编程模型所提供的软件库来访问共享内存区域，如 CUDA 和 OpenCL。这种架构叫作异构架构，如下图所示，其中应用可以在单个地址空间中创建数据结构，并将任务发送给适合于其解析的设备硬件。多个处理任务可以在相同区域中安全地操作以避免数据一致性问题，这要归功于原子操作。因此，虽然 CPU 与 GPU 之间的协同效率不高，但借助这种新型架构，我们可以优化它们之间的交互以及并行应用的性能。



异构架构模式

并行编程模型

并行编程模型是硬件与内存架构的一种抽象。实际上，这些模型并非特定的，也没有指代特定类型的机器或是内存架构。它们可在任何类型的机器上实现（至少理论上如此）。相比于之前的划分来说，这些编程模型位于更高的层级上，表示软件必须被实现为执行一种并行计算的方式。每种模型都有与其他处理器共享信息的方式，从而访问内存并划分任务。

并不存在最好的编程模型，只有最适合程序员所要解决的问题的模型。最广泛使用的并行编程模型有：

- 共享内存模型

- 多线程模型
- 分布式内存 / 消息传递模型
- 数据并行模型

接下来介绍这些模型。后面的章节将会给出更为精确的说明，那时我们将会介绍用于实现它们的适合的 Python 模块。

共享内存模型

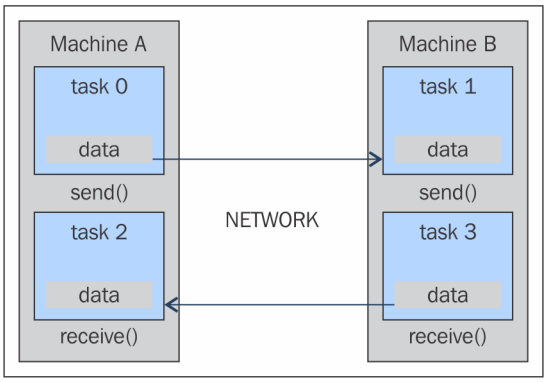
在该模型中，任务共享单个共享的内存区域，对共享资源的访问（读写数据）是异步的。有机制可以让程序员控制对共享内存的访问，比如锁或信号量。该模型的优势在于程序员不必清楚任务间的通信。从性能角度来说，该模型的一个严重缺点是使理解与管理数据的局部性变得更为困难；让数据成为使用它的处理器的局部数据可以减少内存访问、缓存刷新，以及多个处理器使用相同数据时所产生的总线流量。

多线程模型

在该模型中，一个进程可以拥有多个执行流，比如，先创建一个顺序部分，随后创建一系列任务并行执行。通常，这种模型会用在共享内存架构中。因此，对于我们来说，管理线程间的同步就是非常重要的事情了，因为这些线程会操作共享内存，程序员必须防止多个线程同时更新相同的位置。现代的 CPU 在软件与硬件上都是多线程的。Posix 线程就是软件多线程实现的经典例子。英特尔的超线程技术实现了硬件的多线程，如果一个线程停下来或是等待 I/O 操作，那么它会切换至另外一个线程。即便数据对齐是非线性的，我们也可以通过该模型实现并行。

消息传递模型

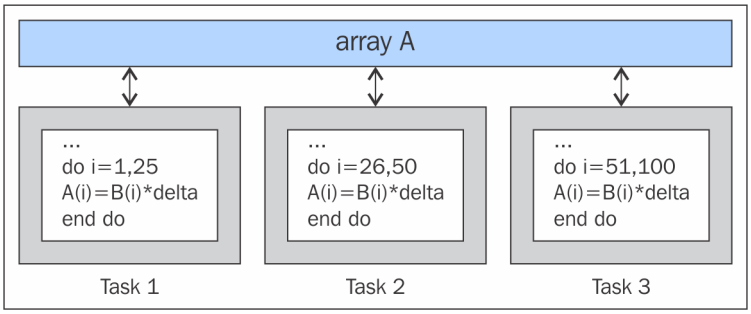
消息传递模型通常用在每个处理器都有自己的内存（分布式内存系统）的场景下。更多的任务可以驻留在同一台物理机或是任意数量的机器上。程序员负责决定并行以及通过消息所进行的数据交换。该并行编程模型的实现需要在代码中用到特殊的软件库。目前已经有了大量的消息传递模型的实现：一些在 20 世纪 80 年代就出现了，不过直到 20 世纪 90 年代中期才形成标准化模型，成为名为 MPI（消息传递接口）的事实上的标准。MPI 模型的设计使用了分布式内存，但却成为并行编程的模型，多个平台也可以使用共享内存机器。消息传递范式模型如下图所示。



消息传递范式模型

数据并行模型

在该模型中，多个任务可以操作相同的数据结构，不过每个任务都只会操作数据的不同部分。在共享内存架构中，所有的任务都可以通过共享内存与分布式内存架构来访问数据，其中的数据结构会被分割并驻留在每个任务的本地内存中。为了实现该模型，程序员需要开发一个程序来指定数据的分布与对齐方式。现代 GPU 在数据对齐的情况下处理速度非常快。数据并行范式模型如下图所示。



数据并行范式模型

如何设计并行程序

针对并行的算法设计基于一系列操作，程序要通过该算法正确地执行任务而不会生成部分或是错误的结果。对于一个算法来说，正确的并行化需要执行的宏观操作有：

- 任务分解

- 任务分配
- 聚集
- 映射

任务分解

这是第一个阶段，在该阶段，软件程序会被分解为任务或是一组指令，接下来在不同的处理器上执行以实现并行化。为了完成这个分解，可以使用以下两种方法。

- **领域分解**：将问题数据进行分解；应用对于使用不同部分数据的所有处理器来说是公共的。当待处理的数据量很大时通常会使用该方法。
- **功能性分解**：在这种情况下，问题会被分解为任务，每个任务都会在所有可用数据上执行特定的操作。

任务分配

在这个步骤中，指定好将任务分发到各个处理器上的机制。这个阶段非常重要，因为它会在各个处理器上建立工作负载的分配机制。在这里，负载均衡尤为重要；实际上，所有处理器必须要不间断地工作，从而避免较长时间的闲置状态。为了做到这一点，程序员需要考虑到系统之间可能存在的异构性，从而在性能更好的处理器上分配更多的任务。最后，要想获得更高的并行效率，要尽可能地限制处理器之间的通信，因为这常常是速度变慢以及资源消耗之源。

聚集

聚集指的是将小任务与大任务合并到一起从而改进性能的过程。如果设计过程的前两个阶段将问题划分为一系列任务，但任务数量远远超过可用的处理器数量，同时计算机并未针对处理大量的小任务而进行特别的设计（有些架构如 GPU 则可以很好地解决这个问题，实际上还会从运行数以百万甚至是数以亿计的任务中获益），那么设计就是极其低效的。一般来说，这是因为任务需要与处理器或是线程进行通信，这样它们才能计算任务。大多数通信的代价不仅与所传输的数据量有关，每次通信操作也都有固定的代价（比如建立 TCP 连接时固有的延迟）。如果任务过小，那么这种固定的代价就很容易使设计变得毫无效率可言。

映射

在并行算法设计过程的映射阶段，我们指定哪个任务将要执行。其目标是将总执行时间降到最低。这里必须要做出权衡，因为两个主要的策略之间经常会彼此冲突：

- 通信频繁的任务应该被放到同一个处理器中来增加局部性。
- 可以并发执行的任务应该被放到不同的处理器中以增强并发性。

这就是映射问题，即 NP 完备。这样，一般来说，并不存在针对问题的多项式时间解决方案。对于相同大小的任务以及很容易确定通信模式的任务来说，映射是很直接的（还可以执行聚集以将映射到相同处理器的任务合并起来）。不过，如果任务的通信模式很难预测或是每个任务的工作量大小千差万别，那么就很难设计出一种高效的映射与聚集模式了。对于这类问题，我们可以通过负载均衡算法在运行期确定聚集与映射策略。最难的是那种在程序执行期间通信量或是任务量发生变化的情况。对于这类问题，可以使用动态的负载均衡算法，它会在执行期间周期性地运行。

动态映射

不同的问题存在着多种负载均衡算法，有全局的，也有局部的。全局算法需要对待执行的计算有全局的掌控，这通常会增加大量的成本。局部算法只依赖于特定于任务本身的信息，相比于全局算法来说，这降低了成本，不过在寻找最优的聚集与映射时情况会变得更糟。不过，虽然映射本身效果更差，但降低的成本却会减少执行时间。如果除了执行开始或是结束外任务之间很少通信，那么我们就可使用任务调度算法来简化处理器空闲时将任务映射到其上的操作。在任务调度算法中会维护一个任务池，任务会被放到这个池中，并由执行者取出。

该模型中存在 3 种常见的方式，下面将会介绍。

管理者 / 执行者

这是一种基本的动态映射模式，所有的使用者都会连接到中心化的管理者。管理者会不断向使用者发送任务并收集结果。这种策略对于数量较少的处理器来说可能是最适合的。可以通过提前获取任务使得通信与计算之间彼此重叠来改进该策略。

层次化的管理者 / 执行者

这是管理者 / 执行者的一个变种，它有一个半分布式的层次；执行者被划分到组里，每个都与自己的管理者相关联。这些组管理者会与中央管理者进行通信（组管理者之间也可以通信），同时执行者会从组管理者那里请求任务。这会将负载分配到几个管理者上，这样如果所有执行者都从同一个管理者请求任务，那么它就可以处理更多数量的处理器了。

去中心化

在这种模式中，一切都是去中心化的。每个处理器会维护自己的任务池，并与其他处理器通信来请求任务。一个处理器选择其他处理器来请求任务的方式是不同的，这是根据问题来决定的。

如何评估并行程序的性能

并行编程的发展使得性能度量与评估并行算法性能的软件成为刚需，这样才可以确定其使

用起来方便与否。实际上，并行计算的关注点在于在相对较短的时间内解决大规模问题。影响该目标的因素有所用的硬件类型、问题的并行度以及所用的并行编程模型等。为了实现这一点，我们引入了基本的概念分析，它会比较从原始序列所获得的并行算法。其性能是通过分析与量化所用的线程数与进程数来达成的。

为了分析这一点，我们引入了一些性能指标：加速、效率与可伸缩性。

阿姆达尔定律提出了并行计算的限制，即由一个串行算法的并行度所决定的，此外，我们还有古斯塔夫定律。

加速

加速是一种度量方式，用于展示以并行方式解决问题所带来的好处。它的定义是，在单个处理元素上解决一个问题所花费的时间 T_S 除以在 p 个相同的处理元素上解决同样问题所花费的时间 T_p。

我们用 $S=\frac{T_S}{T_P}$ 来表示加速。如果 $S=p$，那么加速就是线性的，这表示如果处理器数量增加，那么执行速度也会增加。当然，这是理想情况。当 T_S 是最佳的串行算法执行时间时，加速是绝对的；当 T_S 是单个处理器上并行算法的执行时间时，加速是相对的。

下面再来看看这些条件：

- $S=p$ 是线性的或理想状况下的加速。
- $S<p$ 是真实的加速。
- $S>p$ 是超线性加速。

效率

在理想世界中，拥有 p 个处理元素的并行系统的加速等于 p。不过，这是非常难以实现的。通常情况下，一些时间会浪费在空闲或是通信上。效率是一个性能度量，它会估算相比于在通信与同步上所浪费的时间来说，处理器在解决一个任务时的利用率。

我们将其表示为 E，定义为 $E=\frac{S}{p}=\frac{T_S}{pT_P}$。线性加速算法的 E 值为 1；在其他情况下，E 值要小于 1。如下列出了 E 值的 3 种情况：

- 当 $E=1$ 时，它是线性的。
- 当 $E<1$ 时，它是真实情况的。
- 当 $E<<1$ 时，这是一个并行效率很低的问题。

可伸缩性

可伸缩性指的是并行机器的效能。它是计算能力（执行速度）除以处理器数量的值。如果问题规模变大，同时处理器数量也随之增加，那么性能是不会有损耗的。通过增加不同的因子，可伸缩性系统会保持相同的性能或是改进性能。

阿姆达尔定律

阿姆达尔定律是用于设计处理器与并行算法的广泛使用的定律。它阐释了可获取的最大的加速是由程序中的串行组件所决定的：$S = \frac{1}{1-P}$，其中，$1–P$ 表示程序中的串行组件（非并行部分）。这意味着如果一个程序中 90% 的代码是可并行的，但有 10% 的代码要保持串行，那么可获取的最大的加速就是 9，即便对于无限数量的处理器来说亦如此。

古斯塔夫定律

古斯塔夫定律基于如下假设：

- 在增加问题维度时，其串行部分保持不变。
- 在增加处理器数量时，每个处理器所处理的工作保持不变。

这表明 $S(P) = P - \alpha (P–1)$，其中 P 是处理器数量，S 是加速，α 是任何并行进程中的非并行部分。它与阿姆达尔定律相反，阿姆达尔定律认为单个进程的执行时间是固定的，并且每个进程的并行执行时间会减少。因此，阿姆达尔定律基于固定问题规模这样一个假设；它假设一个问题的全部工作量并不会随着机器规模（即处理器数量）的变化而变化。古斯塔夫定律着重解决了阿姆达尔定律的缺陷，后者并未将解决任务时所涉及的全部计算资源量考虑在内。它表明，设定并行问题解决方案时间的最佳方式是将所有计算资源都考虑进来，基于这一点，它解决了阿姆达尔定律的问题。

Python简介

Python是一门强大、动态的解释型编程语言，广泛使用在各种应用中。其主要特性如下所示：

- 具有清晰且可读性好的语法。
- 具有大量标准库，通过额外的软件模块，我们可以增加数据类型、函数与对象。
- 易于学习的快速开发与调试能力；Python 代码的开发速度最快，可以达到 C/C++ 的 10 倍以上。
- 具有基于异常的错误处理。
- 具有强大的内省功能。
- 具有丰富的文档与软件社区。

可以将 Python 看作一门胶水语言。借助 Python，我们可以开发出更好的应用，因为不同类型的程序员可以在一个项目上协作。比如，在构建科学应用时，C/C++ 程序员可以实现高效的数值算法，而同一项目的科学家则可以编写 Python 程序来测试并使用这些算法。科学家无须学习底层的编程语言，C/C++ 程序员也无须了解这里面涉及的科学内容。

可以通过 https://www.python.org/doc/essays/omg-darpa-mcc-position 了解更多关于这方面的内容。

准备工作

可以从 https://www.python.org/downloads/ 下载 Python。

虽然可以通过 Notepad 或是 TextEdit 来编写 Python 程序，但你会发现使用集成开发环境（IDE）来阅读和编写代码会更加轻松。

有很多专门针对 Python 设计的 IDE，如 IDLE（http://www.python.org/idle）、PyCharm（https://www.jetbrains.com/pycharm/）及 Sublime Text（http://www.sublimetext.com/）等。

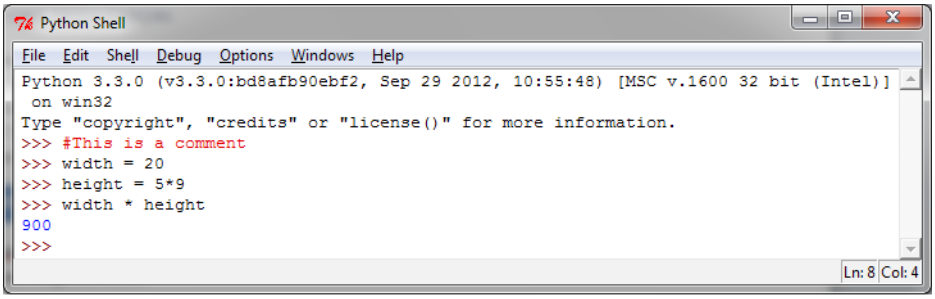
具体实现

下面来看一些非常简单的代码示例以了解 Python 的特性。记住，符号 >>> 表示 Python shell。

- 整型操作：

```
>>> # 这是注释
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

我们只对这第一个示例展示出代码在 Python shell 中的样子，如下图所示。



下面来看看其他基本示例。

- 复杂数字：

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
```

- 字符串操作：

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[-1] # 最后一个字符
'A'
```

- 定义列表：

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> len(a)
4
```

- while 循环：

```
# 斐波那契数列：
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
```

```
2
3
5
8
```

- if 命令：

  首先使用 input() 语句插入一个整数：

```
>>>x = int(input("Please enter an integer here: "))
Please enter an integer here:
```

  然后对插入的数字实现 if 条件：

```
>>>if x < 0:
...       print ('the number is negative')
...elif x == 0:
...       print ('the number is zero')
...elif x == 1:
...       print ('the number is one')
...else:
...       print ('More')
...
```

- for 循环：

```
>>> # 计算字符串长度：
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...      print (x, len(x))
...
cat 3
window 6
defenestrate 12
```

- 定义函数：

```
>>> def fib(n):     # 写出到n的斐波那契数列
...      """ 打印到n的斐波那契数列 """
...      a, b = 0, 1
...      while b < n:
...          print (b),
...          a, b = b, a+b
...
```

```
>>> # 现在调用刚才定义的函数:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

- 导入模块：

```
>>> import math
>>> math.sin(1)
0.8414709848078965

>>> from math import *
>>> log(1)
0.0
```

- 定义类：

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

并行世界中的Python

作为一门解释型语言，Python 是快速的，如果速度很关键，那么它可以轻松地与使用更快的语言所编写的扩展进行交互，如 C 或是 C++。使用 Python 的一种常见方式是用它编写程序的高层次逻辑；Python 解释器是用 C 编写的，叫作 CPython。该解释器会将 Python 代码转换为一种名为 Python 字节码的中间语言，它类似于汇编语言，不过包含了高层次的指令。当运行 Python 程序时，所谓的计算循环会将 Python 字节码转换为特定于机器的操作。解释器的使用对于代码编写与调试来说都有好处，不过程序的执行速度却成为一个问题。一种解决方案是通过第三方包来提供的，程序员编写 C 模块，然后在 Python 中将其导入进来。另一种解决方案是使用即时（Just-in-Time）Python 编译器，它是 CPython 的一种替代方案，比如 PyPy 实现就优化了代码生成和 Python 程序的执行速度。本书将会介绍该问题的第三种解决方案；实际上，Python 提供了能够从并行中获益的特别模块。后续章节将会介绍这些模块，它们使用了

并行编程范式。

不过，本章将会介绍线程与进程这两个基本概念，以及它们在 Python 编程语言中的使用。

进程与线程介绍

进程指的是应用的执行实例，比如，双击桌面上的 Internet 浏览器图标就会开启一个运行该浏览器的进程。线程指的是一个活动的控制流，一个进程中可以同时存在多个活动的线程。术语“流控制”指的是机器指令的顺序执行。此外，一个进程可以包含多个线程，因此启动浏览器时，操作系统就会创建一个进程并开始执行该进程的主线程。每个线程都可以与其他进程或是线程并行独立地执行一组指令（典型的是一个函数）。不过，对于同一个进程中的不同活动线程来说，它们共享地址空间与数据结构。有时，我们也会将线程叫作轻量级进程，这是因为它拥有与进程相同的很多特性，特别的，作为一个顺序的控制流可与其他顺序流并行执行。术语“轻量”指的是线程的实现要比进程轻松得多。不过，与进程不同的是，多线程可以共享很多资源，特别是可以共享地址空间以及数据结构。

下面重述一遍：

- 一个进程可以包含多个并行线程。
- 通常情况下，对于 CPU 资源来说，由操作系统创建和管理的线程要比进程所创建和管理的线程成本低。线程用于小型任务，而进程则用于更为重量级的任务——基本上来说是应用的执行。
- 同一个进程中的线程共享地址空间与其他资源，而进程之间则是彼此独立的。

在深入探索 Python 通过线程与进程来进行并行管理的特性与功能之前，我们先来看看 Python 编程语言是如何使用这两个功能的。

开始在Python中使用进程

在一般的操作系统中，每个程序都会运行在自己的进程内。通常，我们通过双击程序图标或是从菜单中选择来启动一个程序。这里，我们只演示如何从 Python 程序中启动一个新的程序。进程拥有自己的空间地址、数据栈与其他辅助数据来追踪其执行；操作系统会管理所有进程的执行，通过调度程序来管理对于系统中计算资源的访问。

准备工作

在第一个 Python 应用中，我们来安装 Python 语言。

请访问 `https://www.python.org/` 来获取最新版的 Python。

具体实现

为了执行这个示例，需要输入如下两个程序：

- called_Process.py
- calling_Process.py

可以使用 Python IDE (3.3.0) 编辑这些文件：

called_Process.py 文件的代码如下所示：

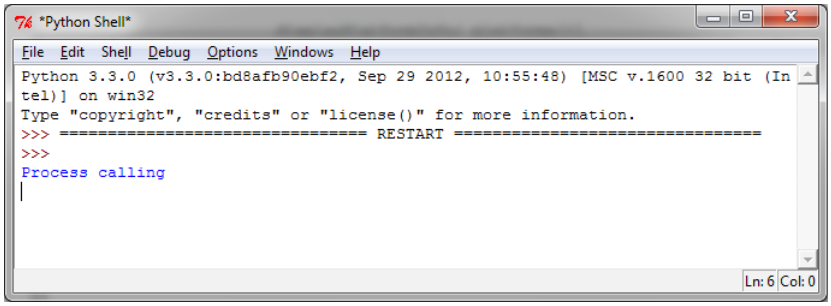
```
print ("Hello Python Parallel Cookbook!!")
closeInput = raw_input("Press ENTER to exit")
print "Closing calledProcess"
```

calling_Process.py 文件的代码如下所示：

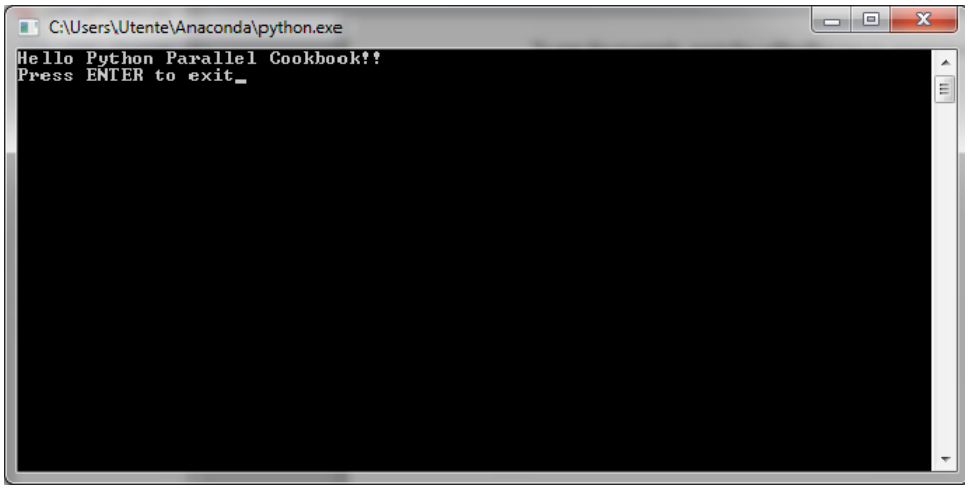
```
##导入如下模块
import os
import sys

##下面是将要执行的代码
program = "python"
print("Process calling")
arguments = ["called_Process.py"]

##调用called_Process.py脚本
os.execvp(program, (program,) + tuple(arguments))
print("Good Bye!!")
```

要想运行该示例，请使用 Python IDE 打开 calling_Process.py 程序，然后按下键盘上的 F5 键。

你会在 Python shell 中看到如下输出。



同时，操作系统提示如下所示。



这里有两个进程在运行，要想关闭操作系统提示，只需按下键盘上的回车键即可。

实例精解

在上述示例中，`execvp` 函数启动了一个新进程，并替换了当前的进程。注意，“Good Bye”消息永远不会被打印出来。相反，它会在标准路径中搜索名为 `called_Process.py` 的程序，并将第 2 个参数元组作为独立参数，将其内容传递给该程序，然后使用当前的环境变量集来运行它。`called_Process.py` 中的指令 `input()` 只用于管理操作系统提示的关闭。在这个专门针对基于进程的并行中，我们最终看到了如何通过多处理 Python 模块来管理更多进程的并行执行。

开始在Python中使用线程

如上一节所述，基于线程的并行是编写并行程序的标准方式。不过，Python 解释器并非完全是线程安全的。为了支持多线程的 Python 程序，我们会使用名为*全局解释器锁*（Global Interpreter Lock）的全局锁。这意味着在同一时刻只有一个线程会执行 Python 代码；在一小段时间后或是当一个线程所做的事情需要花费一段时间时，Python 会自动切换至下一个线程。GIL 并不足以避免程序中的问题。如果多个线程尝试访问同一个对象数据，那么它可能处于不一致的状态下。

在该攻略中，我们将会介绍如何在 Python 程序中创建单个线程。

具体实现

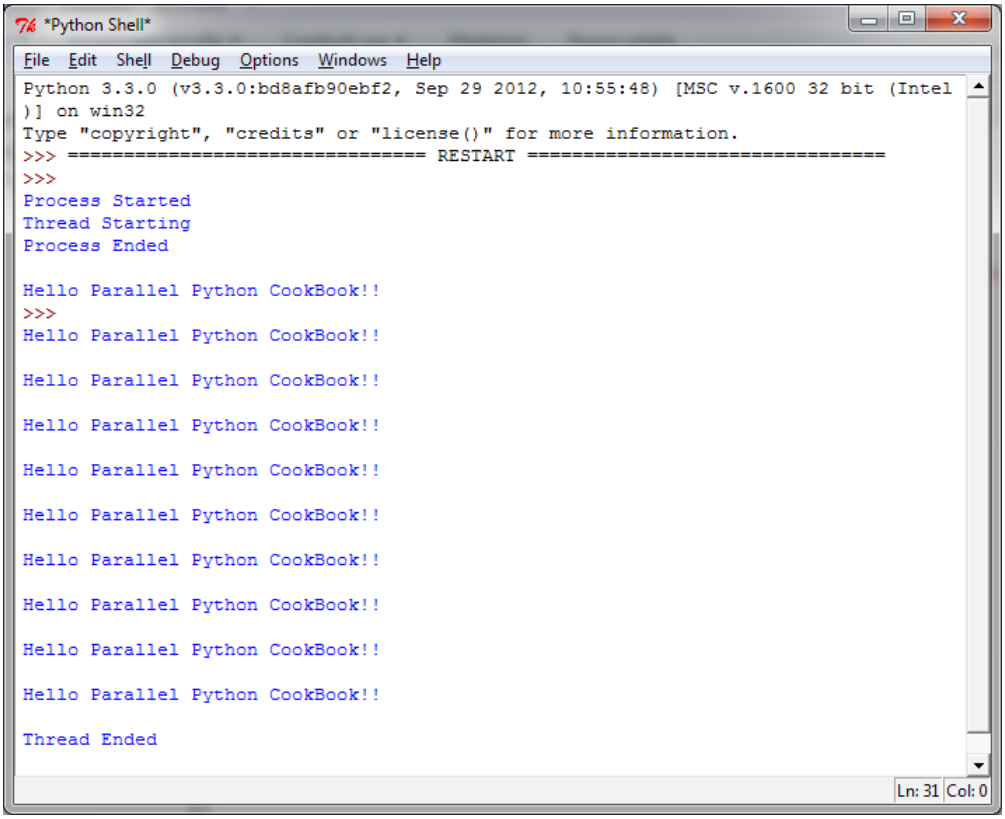
为了执行这个示例，我们需要编写一个名为 helloPythonWithThreads.py 的程序：

```
##要想使用线程，需要通过如下代码导入Thread:
from threading import Thread

##使用sleep函数让线程“睡眠”
from time import sleep

##要想在Python中创建线程，需要让类像线程一样工作
##鉴于此，应该让类继承自Thread类
class CookBook(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.message = "Hello Parallel Python CookBook!!\n"

##如下方法只会打印出消息
    def print_message(self):
        print (self.message)

##run方法会将消息打印10次
    def run(self):
        print ("Thread Starting\n")
        x=0
        while (x < 10):
            self.print_message()
            sleep(2)
            x += 1
        print ("Thread Ended\n")


#开启主进程
print ("Process Started")
```

```
# 创建HelloWorld类的一个实例
hello_Python = CookBook()

# 打印消息，启动线程
hello_Python.start()

# 终止主进程
print ("Process Ended")
```

要想运行该示例，请在 Python IDE 中打开 `calling_Process.py` 程序，然后按下键盘上的 F5 键。

你会在 Python shell 中看到如下图所示的输出。



实例精解

虽然主程序已经执行完毕，但线程还会每隔两秒继续打印消息。该示例演示了何为线程——在一个父进程中做某件事情的子任务。

在使用线程时，关键是要确保永远不要让任何线程在后台运行。这是一种非常差劲的编程方式，在编写大型的应用时会让你问题缠身。

2
基于线程的并行

本章主要内容：

- 如何使用 Python 的线程模块
- 如何定义线程
- 如何确定当前的线程
- 如何在子类中使用线程
- 使用 Lock 与 RLock 实现线程同步
- 使用信号量实现线程同步
- 使用条件实现线程同步
- 使用事件实现线程同步
- 如何使用 with 语句
- 使用队列实现线程通信
- 评估多线程应用的性能
- 多线程编程的关键

介绍

目前，在软件应用中，使用最为广泛的并发管理编程范式是基于多线程的。一般来说，应用是由单个进程所启动的，这个进程又会被划分为多个独立的线程，这些线程表示不同类型的活动，它们并行运行，同时又彼此竞争。

虽然这种编程风格会导致使用上的缺陷以及需要解决一些问题，不过使用了多线程机制的现代应用依旧广泛存在。

实际上，目前所有的操作系统都支持多线程，几乎所有的编程语言都提供了通过线程来实现并发应用的机制。

因此，多线程编程毫无疑问是实现并发应用的一个上佳选择。不过，它并非唯一之选——还有其他一些方案，其中一些方案的性能会更好。

线程是一个独立的执行流，系统中的多个线程可以并行及并发执行。多个线程可以共享数据与资源，利用了所谓的共享信息空间。线程与进程的具体实现取决于应用所运行的操作系统，不过一般来说，线程位于进程内，同一进程中的不同线程共享一些资源。与之相反，不同的进程则不会共享它们的资源。

一个线程主要由 3 个元素构成：程序计数器、寄存器与栈。同一个进程中的线程之间所共享的资源有数据与操作系统资源。类似于进程，线程也有自己的执行状态，并且可以彼此同步。一个线程的执行状态可以分为就绪、运行中与阻塞。一个典型的线程应用当然是应用软件的并行，特别是要利用现代的多核处理器，其中每个核心都会运行一个线程。相比于进程来说，使用线程的好处在于性能方面，进程之间的上下文切换成本要比同一进程中的线程之间的切换成本高很多。

多线程编程更倾向于使用共享信息空间来实现线程之间的通信。这种选择使得多线程编程的主要问题变成了对该空间的管理。

使用Python的线程模块

Python 是通过 Python 标准库所提供的 `threading` 包来管理线程的。该模块提供了一些非常有趣的特性，使得基于线程的开发变得简单很多；实际上，线程模块提供了几种非常容易实现的同步机制。

线程模块的主要组件有：

- 线程对象
- Lock 对象
- RLock 对象
- 信号量对象
- 条件对象
- 事件对象

在下面的攻略中，我们将会通过不同的示例来介绍线程库所提供的各种特性。对于这些示例来说，我们会使用 Python 3.3 版本（不过使用 Python 2.7 也没问题）。

如何定义线程

使用线程最简单的方式是通过一个目标函数来实例化它，然后调用 `start()` 方法使其开始工作。Python 模块 `threading` 提供了 `Thread()` 方法用于在不同的线程中运行进程与函数：

```
class threading.Thread(group=None,
                       target=None,
                       name=None,
                       args=(),
                       kwargs={})
```

在上述代码中，下列项的含义如下所示。

- group：这是 group 的值，应该为 None；这是一个保留参数，供未来实现所用。
- target：这是一个在启动一个线程活动时将会执行的函数。
- name：线程的名字；在默认情况下，形式为 Thread-N 的唯一的名字会赋给它。
- args：这是传给目标的一个参数元组。
- kwargs：这是一个关键字参数字典，供目标函数所用。

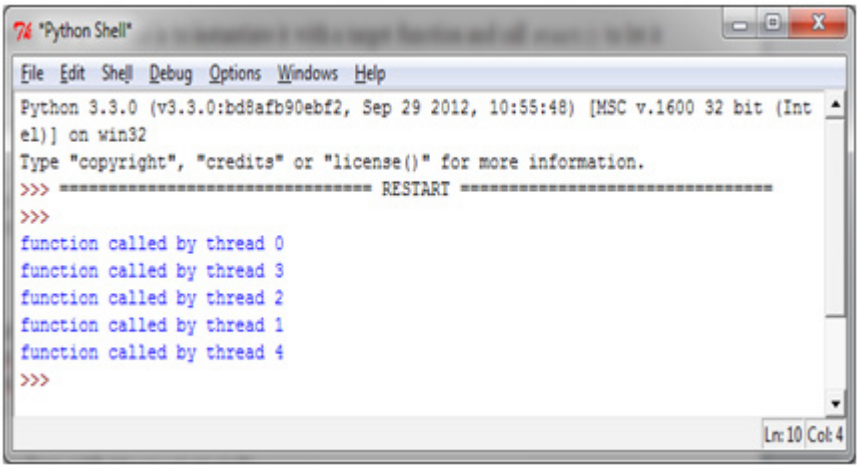
创建一个线程并将参数（告诉线程该做什么事情）传递给它是非常有用的。该示例传递了一个数字，它是线程号，然后打印出结果。

具体实现

下面来看看如何通过线程模块来定义线程，对于该示例来说，寥寥几行代码足矣：

```
import threading

def function(i):
    print ("function called by thread %i\n"  %i)
    return

threads = []
for i in range(5):
    t = threading.Thread(target=function , args=(i,))
    threads.append(t)
    t.start()
    t.join()
```

上述代码的输出如下图所示。



还应该指出的是，可以通过其他方式来获得输出；实际上，多个线程会同时向标准输出打印结果，因此输出顺序是无法确定的。

实例精解

要想导入线程模块，只需使用如下 Python 命令：

```
import threading
```

在主程序中，我们通过 Thread 对象和一个名为 function 的目标函数实例化了一个线程。此外，还将一个参数传递给了该函数，它包含在输出消息中：

```
t = threading.Thread(target=function , args=(i,))
```

直到调用 start() 方法时线程才会开始运行，join() 方法会导致调用线程等待，直到它执行完毕：

```
t.start()
t.join()
```

如何确定当前的线程

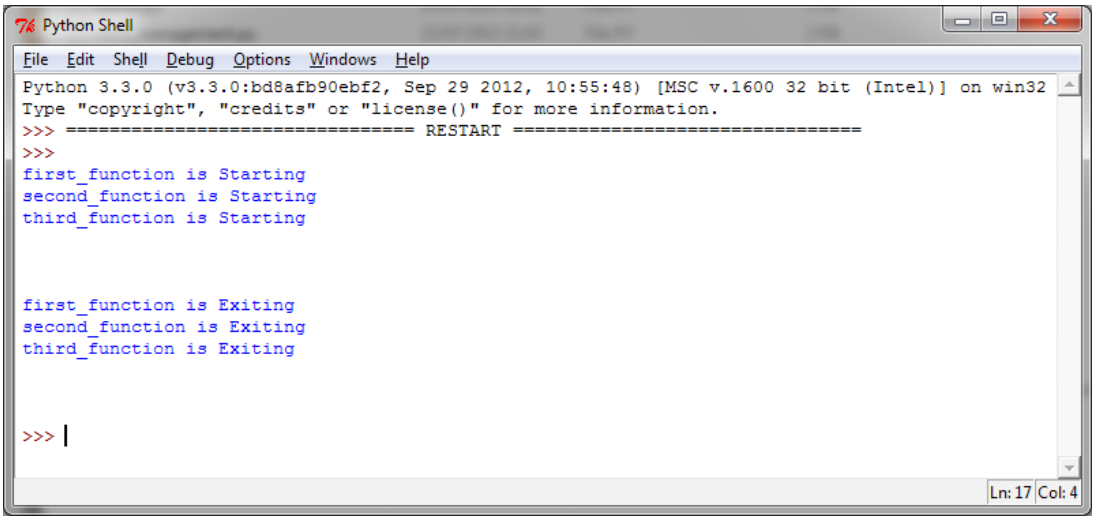
使用参数来标识或是命名线程有些麻烦且不必要。每个线程实例的名字都有一个默认值，并且在创建线程时可以修改这个值。服务端进程可能会有多个服务线程，它们处理不同的操作，这时对线程进行命名就很有用了。

具体实现

为了确定哪个线程正在运行，我们创建 3 个目标函数，并导入 time 模块来暂停执行两秒：

```
import threading
import time

def first_function():
    print (threading.currentThread().getName()+\
           str(' is Starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+\
           str( ' is Exiting \n'))
    return

def second_function():
    print (threading.currentThread().getName()+\
           str(' is Starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+\
           str( ' is Exiting \n'))
    return

def third_function():
    print (threading.currentThread().getName()+\
           str(' is Starting \n'))
    time.sleep(2)
    print (threading.currentThread().getName()+\
           str( ' is Exiting \n'))
    return


if __name__ == "__main__":

    t1 = threading.Thread\
         (name='first_function', target=first_function)
    t2 = threading.Thread\
         (name='second_function', target=second_function)
    t3 = threading.Thread\
         (name='third_function',target=third_function)

    t1.start()
    t2.start()
    t3.start()
```

输出如下图所示。



实例精解

我们使用一个目标函数来实例化一个线程。此外，还传递了名字，这个名字会被打印出来，如果没有定义该名字，那么会使用默认名：

```
t1 = threading.Thread(name='first_function', target=first_function)
t2 = threading.Thread(name='second_function', target=second_function)
t3 = threading.Thread(target=third_function)
```

接下来，对这些线程调用 start() 与 join() 方法：

```
t1.start()
t2.start()
t3.start()
t1.join()
t2.join()
t3.join()
```

如何在子类中使用线程

要想通过线程模块实现新的线程，你需要这样做：

- 定义新的 Thread 类的子类。
- 重写 __init__(self [,args]) 方法来添加额外的参数。
- 接下来，需要重写 run(self [,args]) 方法来实现线程启动后需要做的事情。

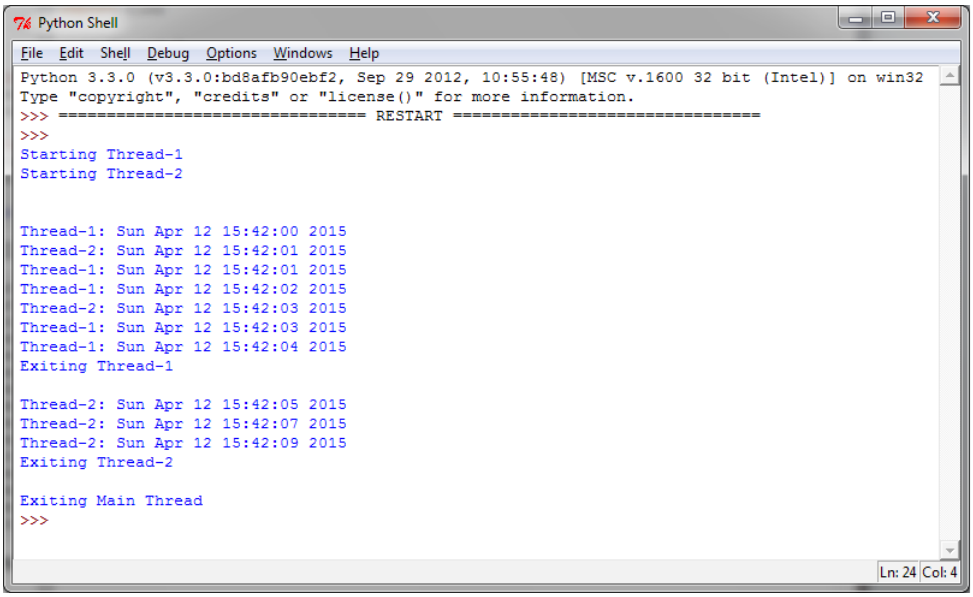
一旦创建好新的 Thread 子类后，你就可以创建它的实例并通过调用 start() 方法来开启新的线程了，start() 方法又会调用 run() 方法。

具体实现

为了在子类中实现线程，我们定义了 myThread 类。它有两个方法，并且需要通过线程参数来重写：

```
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("Starting " + self.name)
        print_time(self.name, self.counter, 5)
        print ("Exiting " + self.name)

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print ("%s: %s" %\
               (threadName, time.ctime(time.time())))
        counter -= 1

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 启动新线程
thread1.start()
thread2.start()
print ("Exiting Main Thread")
```

执行上述代码后，输出结果如下图所示。



实例精解

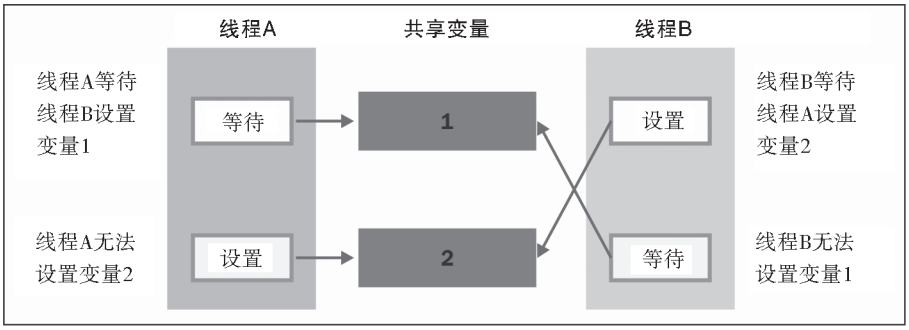
线程模块是创建与管理线程的首选方式。每个线程都由一个类来表示，这个类继承了 Thread 类并重写了 run() 方法。接下来，该方法会成为线程的起始点。在主程序中，我们创建了 myThread 类型的几个对象；当 start() 方法被调用后，线程就会开始执行。调用 Thread 类的构造方法是必需的，这样就可以重新定义线程的一些属性了，比如线程的名字与组。调用 start() 方法后，线程就会处于活动状态，并一直持续下去，直到 run() 方法执行完毕或是抛出了未被处理的异常。当所有线程都终止后，程序也就结束了。

join() 命令只是用来处理线程的终止的。

使用Lock与RLock实现线程同步

当属于并发线程的两个或多个操作尝试访问共享内存，并且至少有一个操作能够修改数据的状态时，这时如果没有恰当的同步机制，就会导致竞态条件，并出现不合法的代码执行、Bug 以及意外的行为。解决竞态条件最简单的方式是使用锁。锁的操作很简单；当一个线程想要访问共享内存的某一部分区域时，它必须要在使用前获取到该部分的锁。此外，在完成操作后，线程必须要释放掉之前获取的锁，这样共享内存的那部分才可以被其他想要使用的线程所用。按照这种方式，显然避免竞态是非常关键的，因为对于线程来说，它对锁的需求要求在某

个给定的时刻，只有一个线程能够使用共享内存的这部分。虽然很简单，但锁是可以正常工作的。不过在实际情况下，我们会看到这种方式常常会导致死锁的发生。死锁的出现是因为不同的线程都持有锁；由于这些锁阻止了线程访问资源，因此无法继续执行操作。死锁的示意图如下图所示。



死锁

出于简单的目的，我们假设有这样一种情况，有两个并发线程（线程 A 与线程 B），同时还有两个资源（1 与 2）。假设线程 A 需要资源 1，线程 B 需要资源 2。在这种情况下，这两个线程都会请求它们自己的锁，一切都可以顺畅地进行下去。不过，如果随后在释放锁之前，线程 A 需要资源 2 的锁，线程 B 需要资源 1 的锁，这对于两个处理来说都是必要的。由于两个资源都被上锁了，因此这两个线程会被阻塞，并相互等待，直到被占用的资源释放为止。这可谓是死锁的最典型的情况。这表示通过使用锁来实现同步，一方面你可以访问到共享内存，另一方面它可能在某些情况下具有破坏性。

在该攻略中，我们介绍了名为 `lock()` 的 Python 线程同步机制。可以通过它将某一时刻对共享资源的访问限定在单个线程或是单个类型的线程上。在访问程序的共享资源前，线程必须要先获取到锁，接下来还必须要能让其他线程访问到相同的资源。

具体实现

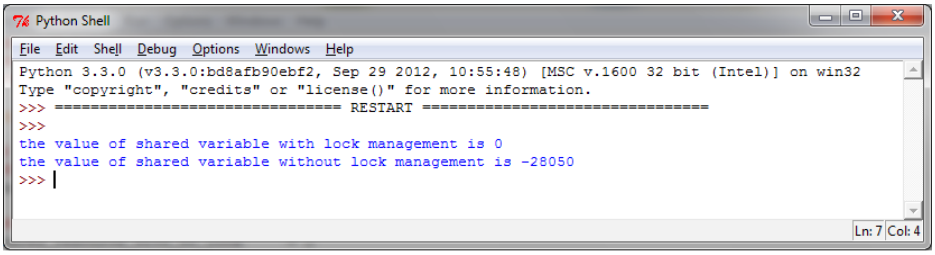
如下示例演示了如何通过 `lock()` 机制来管理线程。代码中有两个函数，分别是 `increment()` 与 `decrement()`。前者会增加共享资源的值，后者则会减少其值，每个函数都被插入到适合的线程中。此外，每个函数都有一个循环，分别用来重复增加或是减少共享资源的值。我们想要确保通过对共享资源的恰当管理，让执行的结果等于共享变量的值，该共享变量的初始值为 0。

示例代码如下所示，代码中所用的每个特性都添加了注释：

```
import threading

shared_resource_with_lock     = 0
shared_resource_with_no_lock     = 0
COUNT = 100000
shared_resource_lock = threading.Lock()


####锁管理##
def increment_with_lock():
    global shared_resource_with_lock
    for i in range(COUNT):

        shared_resource_lock.acquire()
        shared_resource_with_lock += 1
        shared_resource_lock.release()

def decrement_with_lock():
    global shared_resource_with_lock
    for i in range(COUNT):
        shared_resource_lock.acquire()
        shared_resource_with_lock -= 1
        shared_resource_lock.release()

####没有锁管理##
def increment_without_lock():
    global shared_resource_with_no_lock
    for i in range(COUNT):
        shared_resource_with_no_lock += 1

def decrement_without_lock():
    global shared_resource_with_no_lock
    for i in range(COUNT):
        shared_resource_with_no_lock -= 1

####主程序
if __name__ == "__main__":
    t1 = threading.Thread(target = increment_with_lock)
    t2 = threading.Thread(target = decrement_with_lock)
    t3 = threading.Thread(target = increment_without_lock)
    t4 = threading.Thread(target = decrement_without_lock)
    t1.start()
    t2.start()
```

```
    t3.start()
    t4.start()
    t1.join()
    t2.join()
    t3.join()
    t4.join()
    print ("the value of shared variable with lock management is %s"\
           %shared_resource_with_lock)
    print ("the value of shared variable with race condition is %s"\
           %shared_resource_with_no_lock)
```

下图所示的是程序运行后的结果。



如你所见，借助恰当的管理与锁指令，我们得到了正确的结果。注意，没有使用锁管理时的共享变量的结果与其不同。

实例精解

主方法中有如下代码：

```
t1 = threading.Thread(target = increment_with_lock)

t2 = threading.Thread(target = decrement_with_lock)
```

接下来启动线程：

```
t1.start()
t2.start()
```

然后是线程连接：

```
t1.join()
t2.join()
```

在 increment_with_lock() 与 decrement_with_lock() 函数中，你会看到如何使用锁管理。当需要访问资源时，调用 acquire() 来持有锁（如有必要，需要等待锁的释放），并调用 release() 来释放锁：

```
shared_resource_lock.acquire()
shared_resource_with_lock -= 1
shared_resource_lock.release()
```

下面来总结一下。

- 锁有两种状态：上锁与未上锁。
- 有两种方法可用于操纵锁：acquire() 与 release()。

规则如下：

- 如果状态为未上锁，调用 acquire() 会将状态改为上锁。
- 如果状态为上锁，调用 acquire() 会阻塞，直到其他线程调用 release() 为止。
- 如果状态为未上锁，调用 release() 会导致 RuntimeError 异常。
- 如果状态为上锁，调用 release() 会将状态改为未上锁。

知识扩展

虽然理论上可以平稳运行，但锁不仅会导致严重的死锁情况的出现，对于应用来说还会产生其他负面影响。这是一种保守的做法，它本身经常会引入一些不必要的成本；它还会对代码的可伸缩性及可读性产生影响。此外，锁的使用还会与对多个进程所共享的内存的优先级访问需求产生冲突。最后，从实践的角度来看，使用了锁的应用在查错（调试）时会遇到很多困难。因此，我们需要使用其他方法来确保对共享内存的同步访问并避免竞态条件。

使用RLock实现线程同步

如果只想让获取锁的线程来释放锁，那就需要使用一个 RLock() 对象。类似于 Lock() 对象，RLock() 对象也有两个方法：acquire() 与 release()。如果希望在类外面能实现线程安全的访问，同时又使用类里面相同的方法，这时 RLock() 就非常有用了。

具体实现

在示例代码中，我们引入了 Box 类，它拥有 add() 与 remove() 方法，并提供了对 execute() 方法的访问。这样我们就可以执行添加或者删除条目的动作。对 execute() 方法的访问是通过 RLock() 来管理的：

```
import threading
import time
class Box(object):
    lock = threading.RLock()
    def __init__(self):
        self.total_items = 0
```

```
    def execute(self,n):
        Box.lock.acquire()
        self.total_items += n
        Box.lock.release()
    def add(self):
        Box.lock.acquire()
        self.execute(1)
        Box.lock.release()
    def remove(self):
        Box.lock.acquire()
        self.execute(-1)
        Box.lock.release()

## 这两个函数在单独的线程中运行n次
## 并调用Box的方法

def adder(box,items):
    while items > 0:
        print ("adding 1 item in the box\n")
        box.add()
        time.sleep(5)
        items -= 1

def remover(box,items):
    while items > 0:
        print ("removing 1 item in the box")
        box.remove()
        time.sleep(5)
        items -= 1

## 主程序构建出一些线程并确保其
## 可以正常工作
if __name__ == "__main__":
    items = 5
    print ("putting %s items in the box " % items)
    box = Box()
    t1 = threading.Thread(target=adder,args=(box,items))
    t2 = threading.Thread(target=remover,args=(box,items))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print ("%s items still remain in the box " % box.total_items)
```

实例精解

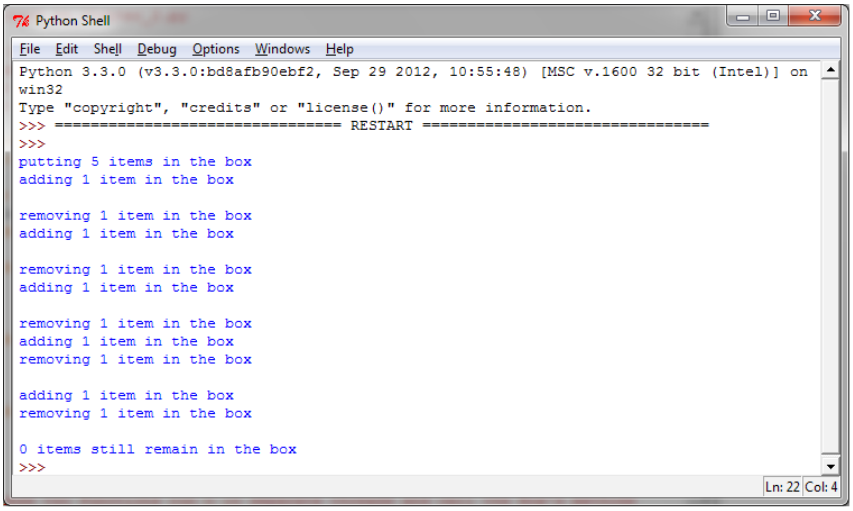
在主程序中，我们重复了上一个示例的内容；两个线程 t1 与 t2 伴随着关联的函数 adder() 与 remover()。当条目的数量大于 0 时，函数会执行。对 RLock() 的调用会在 Box 类中执行：

```
class Box(object):
    lock = threading.RLock()
```

两个函数 adder() 与 remover() 分别会与 Box 类中的条目交互，并调用 Box 类的方法 add() 与 remove()。在每个方法调用中都会捕获资源，然后将其释放掉。对于对象 lock() 来说，RLock() 通过 acquire() 与 release() 方法来对资源进行获取与释放；接下来对于每个方法来说，我们都拥有如下函数调用：

```
   Box.lock.acquire()
#...do something
   Box.lock.release()
```

RLock() 对象的示例执行结果如下图所示。



RLock() 对象的示例执行结果

使用信号量实现线程同步

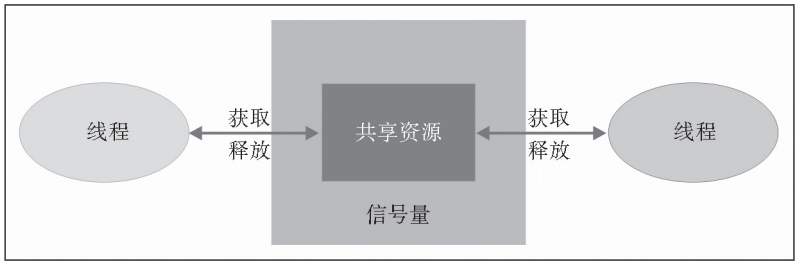
信号量这个概念是由 E. Dijkstra 提出的，并将它首次用在了操作系统中。信号量是一个由操作系统管理的抽象数据类型，用于同步多个线程对共享资源与数据的访问。本质上，信号量

是由一个内部变量构成的，它标识出了对其所关联的资源的并发访问量。

此外，在线程模块中，信号量的操作基于 acquire() 与 release() 这两个函数：

- 当一个线程想要访问与一个信号量所关联的资源时，它必须调用 acquire() 操作，该操作会减少信号量的内部变量值，如果值为非负数，那么它就允许访问资源。如果值为负数，那么线程就会挂起，同时等待另一个线程释放该资源。
- 当一个线程使用完数据或是共享资源，它必须通过 release() 操作释放资源。通过这种方式，信号量的内部变量会增加，信号量队列中的第一个等待线程就可以访问共享资源了。

使用信号量实现线程同步的示意图如下图所示。



使用信号量实现线程同步

虽然初看起来信号量机制并没有什么明显的问题，不过它可以正常工作的前提是等待与信号操作要在原子块中执行。如果不是这样，或是两个操作中的一个停止了，那就会导致我们不期望的情况出现。

假设两个线程同时执行，操作会等待一个信号量，其内部的变量值为 1。另外，再假设当第一个线程将信号量的值由 1 减为 0 时，控制会进入第二个线程，它会将值由 0 减为 –1，并等待，因为内部变量值变为负数了。这时，控制又回到了第一个线程，信号量的值为负数，因此第一个线程也会等待。

因此，尽管信号量可以访问线程，但实际情况却是等待操作没有以原子的形式执行，这就会导致混乱的情况。

准备工作

接下来的代码对问题进行了描述，我们有两个线程，分别分 producer() 与 consumer()，它们共享一个共同资源，这是一个条目（item）。producer() 的任务是生成 item，而 consumer() 线程的任务则是使用所生成的 item。

如果 item 尚未生成，那么 consumer() 线程就需要等待。当 item 生成后，producer() 线

程会通知消费者资源可以使用了。

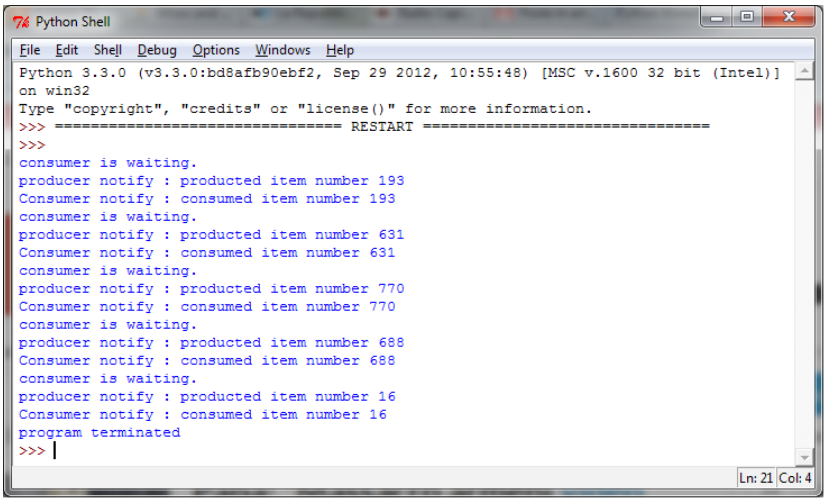
具体实现

在如下代码中，我们使用消费者 - 生产者模型来展示通过信号量实现的同步。当生产者创建一个条目时，它会释放信号量。此外，消费者会获取该信号量并消费共享资源。如下代码展示了通过信号量实现的同步处理。

```
###使用信号量同步线程

import threading
import time
import random


## 可选参数为内部变量counter赋予了初始值，
## 其默认值为1。
## 如果赋的值小于0，那就会导致ValueError异常。
semaphore = threading.Semaphore(0)


def consumer():
    print ("consumer is waiting.")
    ## 获取到信号量
    semaphore.acquire()
    ## 消费者访问共享资源
    print ("Consumer notify : consumed item number %s " %item)


def producer():
    global item
    time.sleep(10)
    ## 创建一个随机数
    item = random.randint(0,1000)
    print ("producer notify : produced item number %s" %item)
    ##释放信号量，将内部的counter值加1。当其值
    ##等于0时，另一个线程就会再次等待它的值变
    ##为大于0，并唤醒该线程。
    semaphore.release()


#主程序
if __name__ == '__main__':
    for i in range (0,5) :
```

```
    t1 = threading.Thread(target=producer)
    t2 = threading.Thread(target=consumer)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
print ("program terminated")
```

下图所示是运行 5 次后的结果。



实例精解

将信号量初始化为 0，我们得到了一个所谓的信号量事件，其唯一的目的在于同步两个或多个线程的计算。这里，一个线程必须同时使用数据或是公共资源：

```
semaphore = threading.Semaphore(0)
```

该操作非常类似于之前介绍的锁机制。producer() 线程创建条目，然后通过调用下面的方法释放资源：

```
semaphore.release()
```

信号量的 release() 方法会增加计数器值，然后通知其他线程。与之类似，consumer() 方法会通过调用如下方法获取数据：

```
semaphore.acquire()
```

如果信号量的计数器值为 0，那么它就会阻塞条件的 acquire() 方法直到收到其他线程的通知。如果信号量的计数器值大于 0，那么它就会减少该值。

最后，所获取到的数据会被打印到标准输出：

```
print ("Consumer notify : consumed item number %s " %item)
```

知识扩展

信号量的一种特别使用方式是互斥。互斥指的是内部变量的初始化值为 1 的信号量，它可以实现对数据与资源访问的互斥操作。

信号量在多线程编程语言中依旧得到了广泛的应用;不过，使用信号量会导致死锁的发生。比如，当线程 t1 在信号量 s1 上执行等待操作时，同时线程 t2 在信号量 s2 上执行等待操作，接下来 t1 在 s2 上执行等待操作，t2 在 s1 上执行等待操作，这就导致了死锁的发生。

使用条件实现线程同步

条件标识了应用中状态的改变。这是一种同步机制，即一个线程等待特定的条件，另一个线程通知它条件已经发生。一旦条件发生，线程就会获取到锁，从而排他性地访问共享资源。

准备工作

阐释该机制的一种好方式是再次看看生产者 - 消费者问题。类 producer 向缓冲区写入数据，直到缓冲区充满为止；只要缓冲区中有数据，类 consumer 就会从缓冲区接收数据（并将数据从缓冲区清除）。类 producer 会通知 consumer 缓冲区不为空，同时 consumer 会告诉 producer 缓冲区没有满。

具体操作

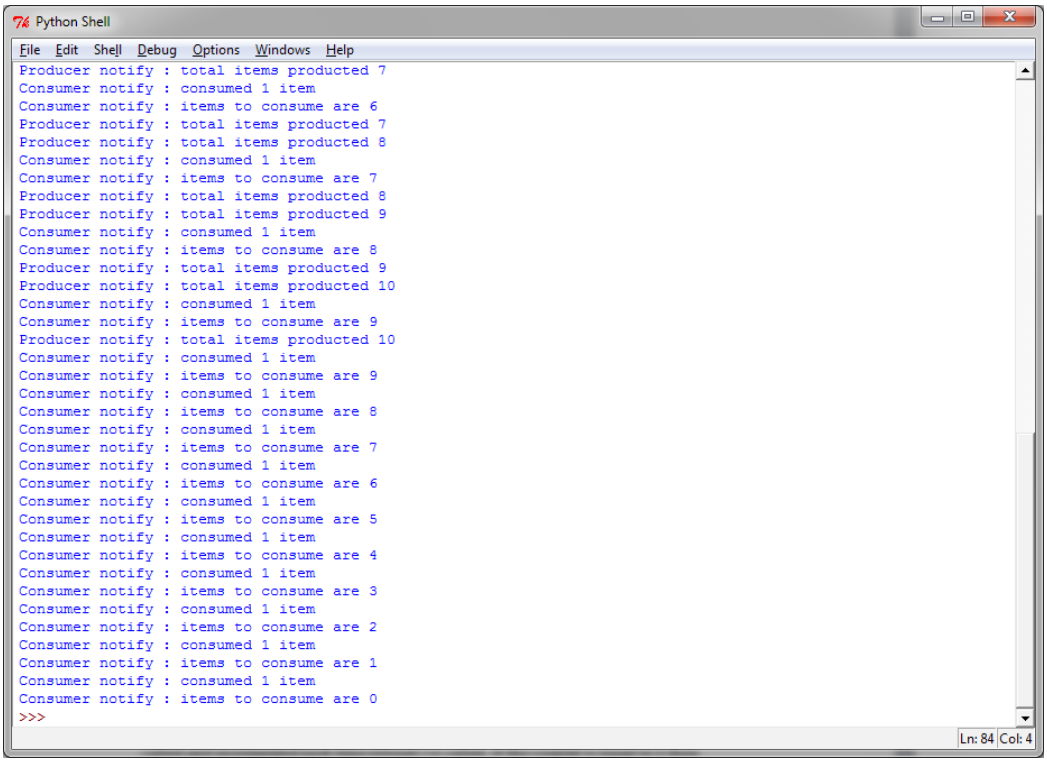
为了演示条件机制，我们继续使用生产者 - 消费者模型：

```
from threading import Thread, Condition
import time

items = []
condition = Condition()

class consumer(Thread):
  def __init__(self):
    Thread.__init__(self)

  def consume(self):
    global condition
    global items
```

```
    condition.acquire()
    if len(items) == 0:
      condition.wait()
      print("Consumer notify : no item to consume")
    items.pop()
    print("Consumer notify : consumed 1 item")
    print("Consumer notify : items to consume are "\
                      + str(len(items)))
    condition.notify()
    condition.release()

  def run(self):
    for i in range(0,20):
      time.sleep(10)
      self.consume()


class producer(Thread):
  def __init__(self):
    Thread.__init__(self)

  def produce(self):
    global condition
    global items

    condition.acquire()
    if len(items) == 10:
      condition.wait()
      print("Producer notify : items producted are "\
                             + str(len(items)))
      print("Producer notify : stop the production!!")
    items.append(1)
    print("Producer notify : total items producted "\
                      + str(len(items)))
    condition.notify()
    condition.release()

  def run(self):
    for i in range(0,20):
      time.sleep(5)
      self.produce()
```

```
if __name__ == "__main__":
        producer = producer()
        consumer = consumer()
        producer.start()
        consumer.start()
        producer.join()
        consumer.join()
```

程序运行一次后的结果如下图所示。



实例精解

类 consumer 会通过列表 `items[]` 获取到共享资源：

```
condition.acquire()
```

如果列表长度为 0，那么消费者就会处于等待状态：

```
if len(items) == 0:
   condition.wait()
```

否则，它会对 items 列表执行 pop 操作：

```
items.pop()
```

这样，消费者的状态就会通知到生产者，共享资源随之被释放：

```
condition.notify()
condition.release()
```

类 producer 获取到共享资源，然后验证列表是否已满（在该示例中，我们将 items 列表的最大数量设为 10）。如果列表已满，那么生产者就会处于等待状态，直到列表被消费为止：

```
condition.acquire()
if len(items) == 10:
   condition.wait()
```

如果列表不满，那就会添加一个 item 进去，然后通知状态并释放资源：

```
condition.notify()
condition.release()
```

知识扩展

了解一下对于条件同步机制的 Python 内核实现是很有趣的。如果没有向内部类 _Condition 的构造方法传递锁对象，那么它就会创建一个 RLock() 对象。此外，在调用 acquire() 与 released() 时会对锁进行管理：

```
class _Condition(_Verbose):
    def __init__(self, lock=None, verbose=None):
        _Verbose.__init__(self, verbose)
        if lock is None:
            lock = RLock()
        self.__lock = lock
```

使用事件实现线程同步

事件是用于线程间通信的对象。一个线程会等待信号，同时另一个线程会发出信号。基本上，事件对象会管理一个内部的标志，可以通过 set() 方法将其设为 true，也可以通过 clear() 方法将其重置为 false。wait() 方法会一直阻塞，直到标志变为 true 为止。

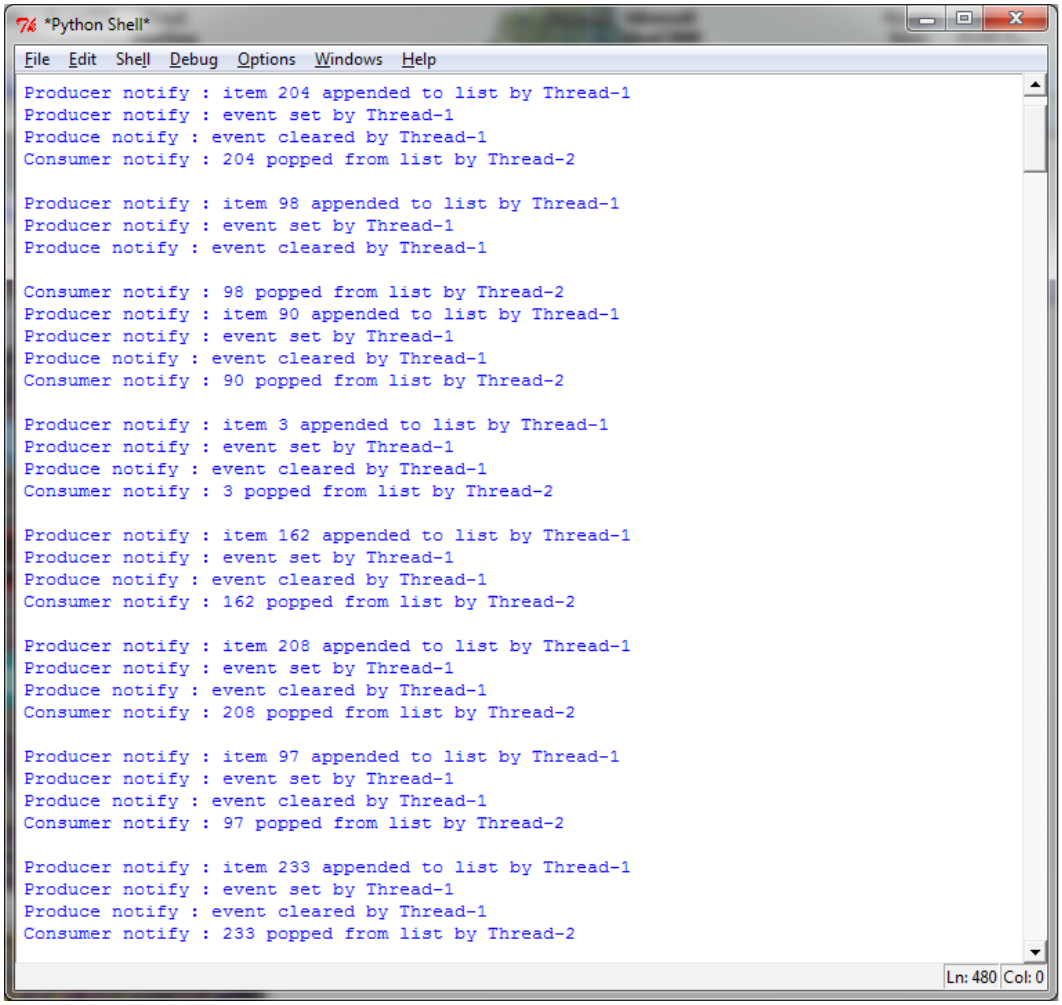
具体操作

为了理解通过事件对象实现的线程同步，我们再来看看生产者 - 消费者问题：

```
import time
from threading import Thread, Event
import random

items = []
event = Event()

class consumer(Thread):
    def __init__(self, items, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

    def run(self):
        while True:
            time.sleep(2)
            self.event.wait()
            item = self.items.pop()
            print ('Consumer notify : %d popped from list by %s'\
                   %(item, self.name))


class producer(Thread):
    def __init__(self, integers, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

    def run(self):
        global item
        for i in range(100):
            time.sleep(2)
            item = random.randint(0, 256)
            self.items.append(item)
            print ('Producer notify : item N° %d appended \
                   to list by %s'\
                   % (item, self.name))
            print ('Producer notify : event set by %s'\
                   % self.name)
            self.event.set()
            print ('Produce notify : event cleared by %s \n'\
                   % self.name)
            self.event.clear()
```

```
if __name__ == '__main__':
    t1 = producer(items, event)
    t2 = consumer(items, event)
    t1.start()
    t2.start()
    t1.join()
        t2.join()
```

下图是程序运行后的输出。t1 线程会向列表附加一个值，接下来设置事件来通知消费者。消费者对 wait() 的调用会停止阻塞，并从列表中获取该整型值。



实例精解

producer 类会通过条目列表与 Event() 函数来初始化。与条件对象示例不同的是，条目列表并非全局的，而是作为一个参数传递进来的：

```
class consumer(Thread):
    def __init__(self, items, event):
        Thread._init__(self)
        self.items = items
        self.event = event
```

在所创建的每个条目的 run 方法中，producer 类会将其附加到条目列表中，然后通知事件。这里涉及两个步骤，第一步如下代码所示：

```
self.event.set()
```

第二步如下所示：

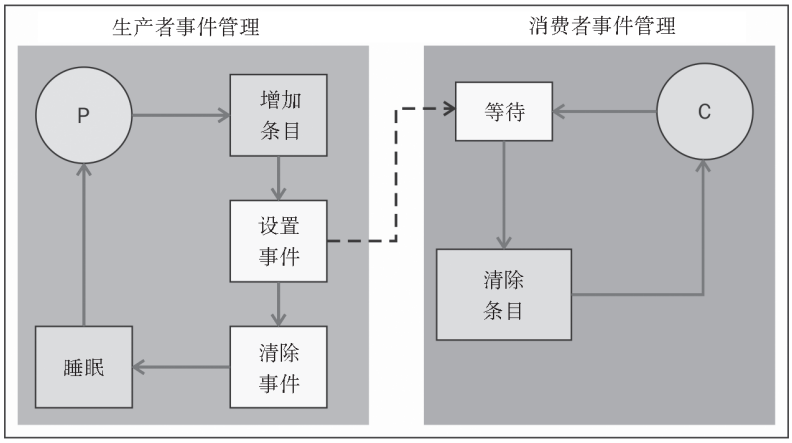
```
self.event.clear()
```

consumer 类由条目列表与 Event() 函数进行初始化。

在 run 方法中，消费者会等待新的条目进行消费。当条目到达时，它会从条目列表中弹出：

```
def run(self):
    while True:
       time.sleep(2)
       self.event.wait()
       item = self.items.pop()
       print ('Consumer notify : %d popped from list by %s' %
              (item, self.name))
```

下图展示了 producer 与 consumer 类之间的所有操作。



使用事件对象实现的线程同步

使用with语句

Python 中的 with 语句是在 Python 2.5 中引入的。当有两个相关的操作需要对一个代码块成对执行时，with 语句的作用就彰显出来了。此外，借助 with 语句，还可以在需要时精确地分配与释放资源；出于这个原因，with 语句又叫作上下文管理器。在线程模块中，acquire() 与 release() 方法所提供的全部对象可以用在 with 语句块中。

因此，如下对象可用作 with 语句的上下文管理器：

- Lock
- RLock
- 条件
- 信号量

准备工作

在该示例中，我们使用 with 语句来测试所有的对象。

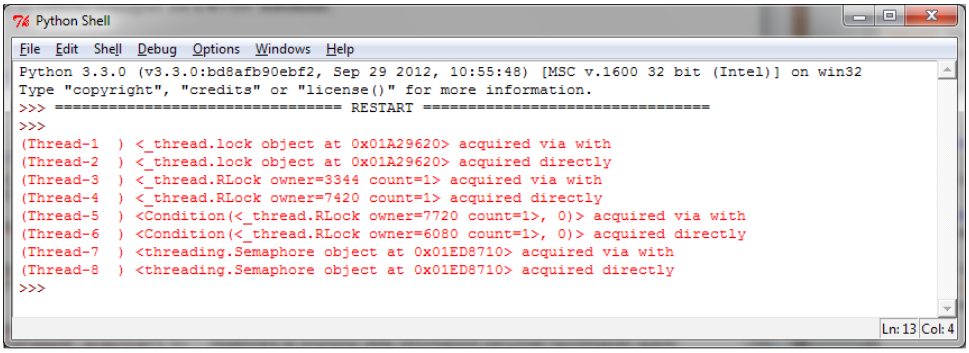
具体操作

该示例演示了 with 语句的基本使用。我们有一个集合，它里面存放的是最重要的同步原语。因此，我们可以通过 with 语句调用其中的每一个来测试：

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',)

def threading_with(statement):
    with statement:
        logging.debug('%s acquired via with'  %statement)

def threading_not_with(statement):
    statement.acquire()
    try:
        logging.debug('%s acquired directly' %statement )
    finally:
        statement.release()
```

```
if __name__ == '__main__':

# 创建一个测试组合
    lock = threading.Lock()
    rlock = threading.RLock()
    condition = threading.Condition()
    mutex = threading.Semaphore(1)
    threading_synchronization_list = \
      [lock, rlock, condition, mutex]

# 在for循环中，调用 threading_with
  与threading_no_with 函数
    for statement in threading_synchronization_list :
        t1 = threading.Thread(target=threading_with,
             args=(statement,))
        t2 = threading.Thread(target=threading_not_with,
             args=(statement,))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
```

下图所示的输出展示了对于每个函数来说，使用 with 语句与不使用 with 语句的结果。



实例精解

在主程序中，我们定义了一个列表，threading_synchronization_list，里面存放的是用于测试的线程通信指令：

```
lock = threading.Lock()
rlock = threading.RLock()
condition = threading.Condition()
```

```
mutex = threading.Semaphore(1)
threading_synchronization_list = \

   [lock, rlock, condition, mutex]
```

定义后，在 for 循环中传递每个对象：

```
for statement in threading_synchronization_list :
        t1 = threading.Thread(target=threading_with,
             args=(statement,))
        t2 = threading.Thread(target=threading_not_with,
             args=(statement,))
```

最后，我们有两个目标函数，其中，threading_with 用于测试 with 语句：

```
def threading_with(statement):
    with statement:
        logging.debug('%s acquired via with' %statement)
```

知识扩展

在如下示例中，我们使用了 Python 的日志支持：

```
logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',)
```

它使用格式化代码 %(threadName) 语句在每条日志消息中加入了线程名。日志模块是线程安全的，因此不同线程所生成的消息在输出中是独立的。

使用队列实现线程通信

如前所述，当线程需要共享数据或是资源时，线程模块会变得非常复杂。我们已经看到了，Python 线程模块提供了很多同步原语，包括信号量、条件变量、事件与锁。虽然存在这么多选择，但使用队列模块可能是一个最佳方式。队列使用起来很容易，并且使得线程编程变得更加安全，因为它们会对单个线程对资源的所有访问进行过滤，并且支持更加整洁且可读性更棒的设计模式。

我们将会介绍如下 4 种队列方法。

- put()：将一个条目添加到队列中。
- get()：从队列中删除并返回一个条目。
- task_done()：每次处理一个条目时都会调用该方法。
- join()：这会导致阻塞，直到所有条目都被处理完为止。

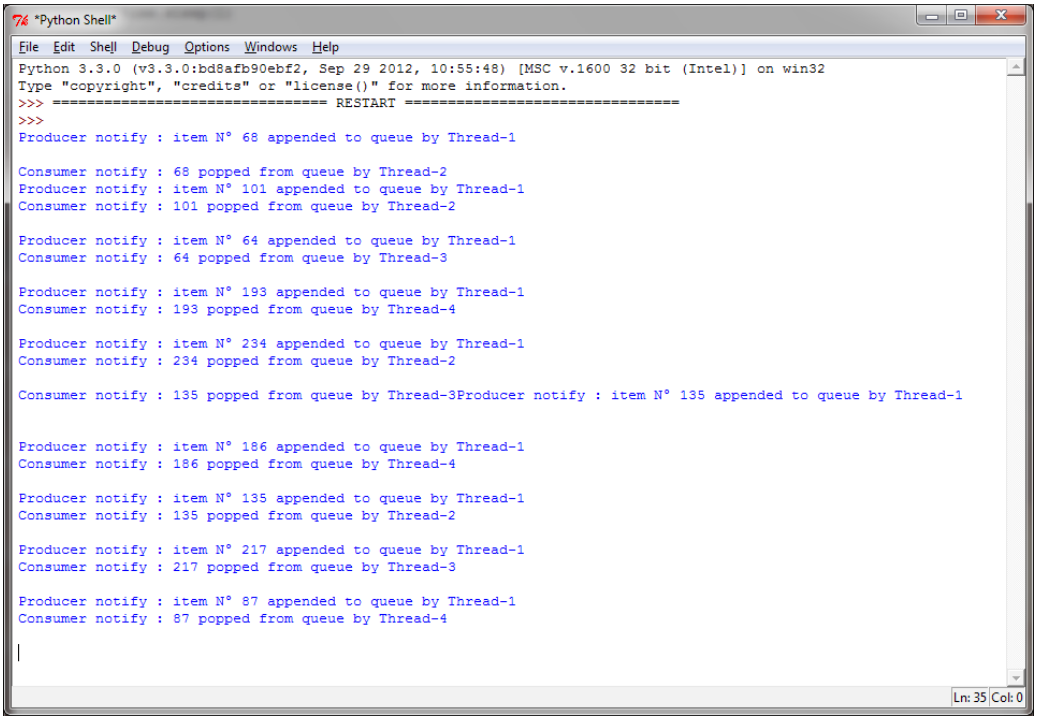
具体操作

在该示例中，我们将会介绍如何同时使用线程模块与队列模块。此外，我们有两个实体，它们会同时尝试共享一个公共资源——队列。代码如下所示：

```
from threading import Thread, Event
from queue import Queue
import time
import random


class producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print ('Producer notify: item N°%d appended to queue by %s
                   \n'\
                   % (item, self.name))
            time.sleep(1)


class consumer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            item = self.queue.get()
            print ('Consumer notify : %d popped from queue by %s'\
                   % (item, self.name))
            self.queue.task_done()


if __name__ == '__main__':
        queue = Queue()
        t1 = producer(queue)
        t2 = consumer(queue)
```

```
    t3 = consumer(queue)
    t4 = consumer(queue)
    t1.start()
    t2.start()
    t3.start()
    t4.start()
    t1.join()
    t2.join()
    t3.join()
    t4.join()
```

代码运行后的输出如下图所示。



实例精解

首先来介绍 producer 类。我们无须传递整数列表，因为使用了队列来存储生成的整数：

```
class producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue
```

producer 类中的线程会在一个 for 循环中生成整数并将其放到队列中：

```
def run(self) :
        for i in range(100):
            item = random.randint(0, 256)
            self.queue.put(item)
```

producer 类使用 Queue.put(item[, block[, timeout]]) 向队列中插入数据。在将数据插入到队列前它会获取到锁。

存在两种可能：

- 如果可选参数 block 为 true，并且 timeout 为 None（这是示例中所用的默认情况），那么就需要阻塞，直到有可用的位置为止。如果 timeout 是一个正数，那么就会阻塞至多 timeout 秒，如果在这期间没有可用的位置，那就会抛出异常。
- 如果 block 为 false，那么在有可用的位置时就会将一个条目放到队列中；否则就会抛出异常（在这种情况下会忽略掉 timeout）。这里的 put() 会检查队列是否已满，接下来在内部调用 wait()，这样生产者就会开始等待。

接下来是 consumer 类。线程会从队列中获取到整数，并调用 task_done() 来标识它已经开始对其进行处理了：

```
def run(self):
        while True:
            item = self.queue.get()
            self.queue.task_done()
```

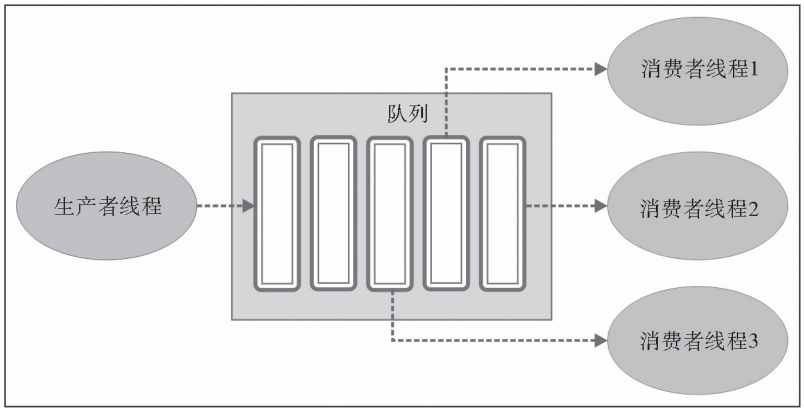
消费者会使用 Queue.get([block[, timeout]])，并在从队列中移除数据前获取到锁。如果队列为空，那么消费者就会进入等待状态。

最后，在主方法中，我们为 producer 类创建了线程 t，为 consumer 类创建了线程 t1、t2 与 t3：

```
if __name__ == '__main__':
        queue = Queue()
        t = producer(queue)
        t1 = consumer(queue)
        t2 = consumer(queue)
        t3 = consumer(queue)

        t.start()
        t1.start()
        t2.start()
        t3.start()
```

```
t.join()
t1.join()
t2.join()
t3.join()
```

producer 类与 consumer 类之间的所有操作总结如下图所示。



使用队列模块实现的线程同步

评估多线程应用的性能

在该攻略中，我们来验证 GIL 的影响，评估多线程应用的性能。如前所述，GIL 是由 CPython 解释器所引入的锁机制。GIL 会在解释器中防止多线程的并行执行。在执行前，每个线程都必须等待 GIL 释放正在运行的线程。实际上，在访问解释器中的任何东西作为栈和 Python 对象实例前，解释器会强制执行中的线程获取到 GIL。这正是 GIL 的目的所在——它会防止不同线程并发访问 Python 对象。GIL 会保护解释器的内存，并确保垃圾收集以正确的方式进行。实际上，如果开发者想以并行执行线程的方式来达到提升性能的目的，那么 GIL 会阻止这样做。如果从 CPython 解释器中删除 GIL，那么线程就能并行执行了。GIL 并不会阻止一个进程在不同的处理器上执行，同一时刻它只允许唯一的线程出现在解释器中。

具体操作

如下代码用来评估一个多线程应用的性能。每个测试都会在 100 次循环迭代中调用一个函数一次。接下来，我们会看到这 100 次调用中速度最快的一个。在 for 循环中，我们调用了 non_threaded 和 threaded 函数。此外，我们迭代测试，增加调用与线程数量。我们测试了 1、2、3 与 4，调用线程最后使用了 8。在非线程执行中，我们根据所用的线程顺序调用了函数相同的次数。为了让事情简单一些，所有执行速度的度量都是由 Python 的 timer 模块来提供的。

该模块用于评估 Python 代码的性能，这些代码只是一些单行的语句。

代码如下所示：

```
from threading import Thread

class threads_object(Thread):
  def run(self):
    function_to_run()

class nothreads_object(object):
  def run(self):
    function_to_run()

def non_threaded(num_iter):
  funcs = []
  for i in range(int(num_iter)):
    funcs.append(nothreads_object())
  for i in funcs:
    i.run()

def threaded(num_threads):
  funcs = []
  for i in range(int(num_threads)):
    funcs.append(threads_object())
  for i in funcs:
    i.start()
  for i in funcs:
    i.join()

def function_to_run():
    pass

def show_results(func_name, results):
  print ("%-23s %4.6f seconds" % (func_name, results))

if __name__ == "__main__":
   import sys
   from timeit import Timer

   repeat = 100
   number = 1
   num_threads = [ 1, 2, 4, 8]
```

```
    print ('Starting tests')
    for i in num_threads:
            t = Timer("non_threaded(%s)"\
                        % i, "from __main__ import non_threaded")
            best_result =\
                        min(t.repeat(repeat=repeat, number=number))
            show_results("non_threaded (%s iters)"\
                       % i, best_result)

            t = Timer("threaded(%s)"\
                        % i, "from __main__ import threaded")
            best_result = \
                        min(t.repeat(repeat=repeat, number=number))
            show_results("threaded (%s threads)"\
                           % i, best_result)

print ('Iterations complete')
```

实例精解

我们一共执行了 3 个测试，对于每个测试来说，都用了不同的函数，修改了定义在示例代码 `function_to_run()` 中的函数代码。

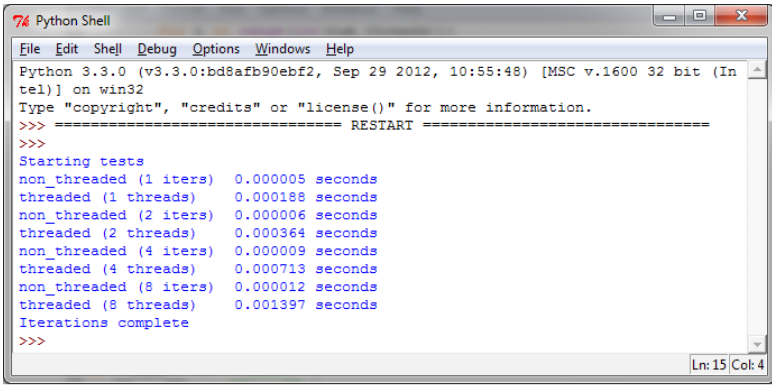
用于测试的机器配置为 Core 2 Duo CPU–2.33Ghz。

第一个测试

在该测试中，我们只是评估空函数：

```
def function_to_run():
    pass
```

下面所示的结果展示了测试的每种机制的开销。



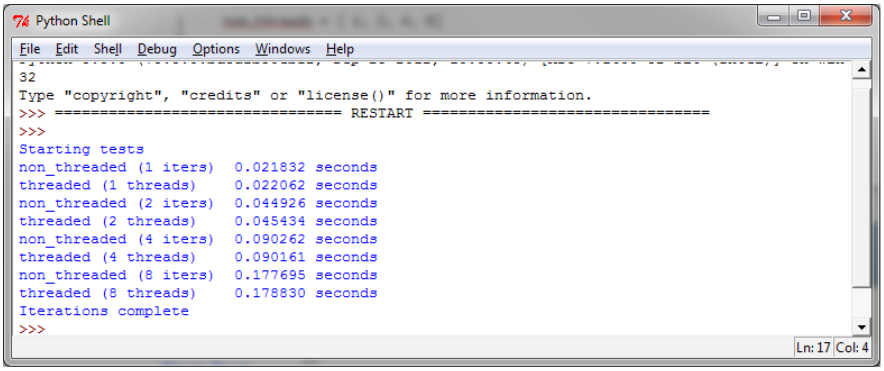
根据结果可以看到线程调用的成本要比不使用线程调用的成本高多少。特别的，我们还会发现增加线程的开销与线程数之间的比例关系；在该示例中，4 个线程的时间为 0.0007143 秒，而 8 个线程的时间为 0.001397 秒。

第二个测试

线程应用的一个典型示例就是对数字的处理。我们通过一个简单的方法来计算斐波那契数列；注意这里是没有共享状态的，只是使用更多的任务来生成数字序列：

```
def function_to_run():
    a, b = 0, 1
    for i in range(10000):
        a, b = b, a + b
```

输出如下图所示。



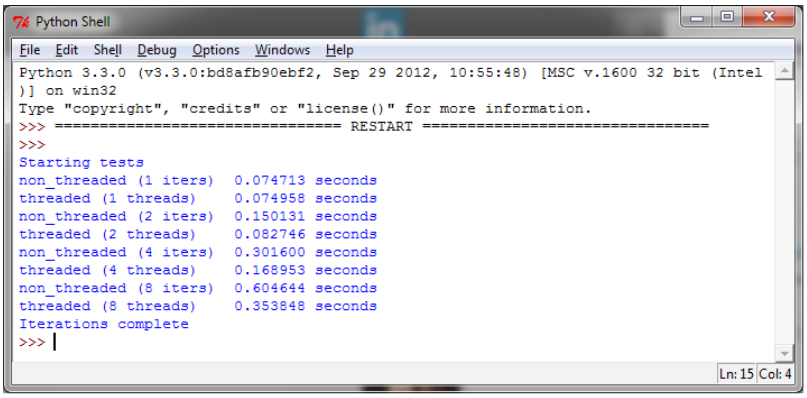
从输出可以看到，增加线程数并未带来什么好处。函数是在 Python 中执行的，由于创建线程与 GIL 的成本，多线程示例永远不可能比非线程示例更快。另外，要记住，在同一时刻，GIL 只允许一个线程访问解释器。

第三个测试

如下测试会从 test.dat 文件中读取一块数据（1Kb）1000 次。待测试的函数如下所示：

```
def function_to_run():
    fh=open("C:\\CookBookFileExamples\\test.dat","rb")
    size = 1024
    for i in range(1000):
        fh.read(size)
```

测试的输出结果如下图所示。



我们看到了在多线程示例中更好的一个结果。特别的，我们注意到相比于 non_threaded，线程执行所花时间为何减半。在实际情况下，我们是不会将线程作为基准的。典型的，将线程放到队列中，并执行其他任务。虽然在某些情况下，多个线程执行同样的函数很有用，但对于并发程序来说却并不常见，除非它会分割输入数据。

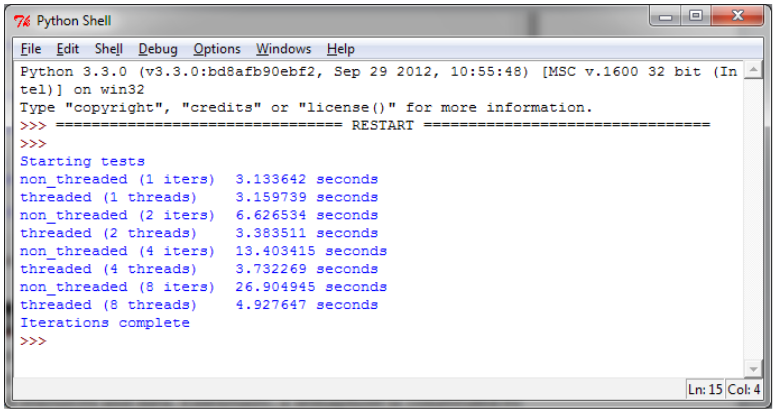
第四个测试

在最后一个示例中，我们使用了 urllib.request，这是一个 Python 模块，用于获取 URL。该模块基于 socket 模块，使用 C 编写并且线程安全。

如下脚本会访问 https://www.packtpub.com/ 主页，并且读取前 1K 字节：

```
def function_to_run():
    import urllib.request
    for i in range(10):
        with urllib.request.urlopen("https://www.packtpub.com/")as f:
            f.read(1024)
```

上述代码的输出结果如下图所示。



如你所见，在 I/O 过程中，GIL 会被释放。多线程执行要比单线程执行快。由于很多应用会在 I/O 中执行一定数量的工作，因此 GIL 并不会阻止程序员创建多线程任务来并发执行以加快执行速度。

知识扩展

我们并未通过增加线程的方式来加快应用的启动时间，而是添加了对并发的支持。比如，创建线程池，然后再重用执行者是非常有益的。这样，我们就可以分割一个大的数据集，并对其不同部分执行相同的函数（生产者 - 消费者模型）。因此，虽然这并非并发应用的典型情况，但这些测试保持了足够的简单性。对于想要使用纯 Python 并希望利用多核硬件架构的应用来说，GIL 会成为一个障碍吗？没错。虽然线程是一种语言构造，但 CPython 解释器却是线程与操作系统之间的桥梁。这也正是 Jython、IronPython 与其他解释器并没有提供 GIL 的原因所在，因为根本没必要，它并未在解释器中重新实现。

3
基于进程的并行

本章主要内容：

- 使用 multiprocessing Python 模块
- 如何生成进程
- 如何对进程进行命名
- 如何在后台运行进程
- 如何杀死进程
- 如何在子类中使用进程
- 如何在进程间交换对象
- 使用队列来交换对象
- 使用管道来交换对象
- 如何同步进程
- 如何管理进程间状态
- 如何使用进程池
- 使用 mpi4py Python 模块
- 点对点通信
- 避免死锁问题
- 使用广播实现聚合通信
- 使用 scatter 函数实现聚合通信
- 使用 gather 函数实现聚合通信
- 使用 AlltoAll 实现聚合通信
- 汇聚操作
- 如何优化通信

介绍

第 2 章介绍了如何通过线程来实现并发应用，本章将会介绍基于进程的方式。特别的，我们的关注点将会放在两个库上：Python 的 multiprocessing 模块与 mpi4py 模块。

Python 的 multiprocessing 库是语言标准库的一部分，它实现了共享内存编程范式，即系统包含一个或多个处理器，它们都可以访问到一个公共内存。

Python 的 mpi4py 库实现了名为消息传递的编程范式，它并不会使用共享资源（这也叫作无共享），并且所有通信都是通过进程间所交换的消息来实现的。

对于这些特性来说，它们与提供内存共享并使用锁或类似的机制来实现互斥操作的通信技术相反。在消息传递代码中，进程是通过 send() 与 receive() 通信原语连接起来的。

在 Python 的 multiprocessing 文档的介绍中清晰地提到了该包的所有功能都需要将主模块导入到子模块中（https://docs.python.org/3.3/library/multiprocessing.html）。

在 IDLE 中是无法将 __main__ 模块导入子模块的，即便将脚本以文件的形式运行也不行。为了得到正确的结果，我们会从命令提示符中运行所有示例：

```
python multiprocessing_example.py
```

这里的 multiprocessing_example.py 是脚本的名字。对于本章所介绍的示例来说，我们将会使用 Python 3.3 版本（Python 2.7 也是可以的）。

如何生成进程

术语“生成（spawn）”指的是通过父进程来创建进程。当然，父进程会以异步的形式继续执行，或是等待，直到子进程执行结束为止。Python 的 multiprocessing 库可以通过如下步骤来生成进程：

1. 构建对象进程。
2. 调用其 start() 方法。该方法会开启进程活动。
3. 调用其 join() 方法。它会等待，直到进程完成其工作并退出为止。

具体操作

该示例展示了如何创建一系列（5 个）进程。每个进程都关联到 foo(i) 函数，其中 i 指的是与进程所关联的 ID：

```
# 生成一个进程：第 3 章：基于进程的并行
import multiprocessing
```

```
def foo(i):
    print ('called function in process: %s' %i)
    return

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=foo, args=(i,))
        Process_jobs.append(p)
        p.start()
        p.join()
```

要想运行该进程并显示出结果，需要打开命令提示符，推荐在包含示例文件（叫作spawn_a_process.py）的目录中打开，然后输入下面的命令：

```
python spawn_a_process.py
```

运行该命令后的输出如下所示：

```
C:\Python CookBook\ Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python spawn_a_process.py
called function in process: 0
called function in process: 1
called function in process: 2
called function in process: 3
called function in process: 4
```

实例精解

正如在本攻略介绍部分所提及的，要想创建对象进程，首先需要通过如下命令导入multiprocessing模块：

```
import multiprocessing
```

接下来在主程序中创建对象进程：

```
p = multiprocessing.Process(target=foo, args=(i,))
```

然后调用start()方法：

```
p.start()
```

对象进程将函数作为参数，子进程会关联到其上（在该示例中，函数叫作foo()）。我们还向函数传递了一个参数，用来指定与进程所对应的那个函数。最后，对所创建的进程调用join()方法：

```
p.join()
```

如果不调用p.join()，那么子进程就会空闲下来但不会终止，接下来就必须要手动杀死它。

知识扩展

这再一次提醒了我们在 main 部分实例化 Process 对象的重要性：

```
if __name__ == '__main__':
```

这是因为所创建的子进程会导入包含了目标函数的脚本文件。接下来，通过在该块中实例化进程对象，可以防止出现无限递归调用实例化的情况。一种有效的解决方法是在另外一个脚本中定义目标函数，然后将其导入命名空间。因此，对于第一个示例来说，我们会这样写：

```
import multiprocessing
import target_function

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process   \
            (target=target_function.function,args=(i,))
         Process_jobs.append(p)
         p.start()
         p.join()
```

下面是 target_function.py 文件的代码：

```
#target_function.py

def function(i):
    print ('called function in process: %s' %i)
    return
```

输出与上一个示例类似。

如何对进程命名

在上一个示例中，我们标识了进程并介绍了如何向目标函数传递变量。不过，为进程关联一个名字会很有用，因为调试应用时将进程标识为可识别的有助于问题的解决。

具体操作

对进程进行命名的过程类似于之前介绍的线程库（参见本书第 2 章）。

在主程序中，我们创建了一个带有名字的进程和一个不带名字的进程。这两个进程都使用了同一个目标函数 foo()：

```
#对进程进行命名：第3章：基于进程的并行
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    time.sleep(3)
    print ("Exiting %s \n" %name)

if __name__ == '__main__':
    process_with_name = \
                      multiprocessing.Process\
                      (name='foo_process',\
                       target=foo)
    process_with_name.daemon = True
    process_with_default_name = \
                               multiprocessing.Process\
                               (target=foo)
    process_with_name.start()
    process_with_default_name.start()
```

要运行该进程，请打开命令提示符并输入如下命令：

```
python naming_process.py
```

下面是运行命令后的输出结果：

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python naming_process.py
Starting foo_process
Starting Process-2
Exiting foo_process
Exiting Process-2
```

实例精解

该操作类似于对线程命名的过程。要想对进程命名，应该提供一个用于标识对象名的参数：

```
process_with_name = multiprocessing.Process
           (name='foo_function', target=foo)
```

在该示例中，我们调用了 foo_function 进程。如果子进程想知道其所属的父进程是谁，那么它需要使用如下语句：

```
name = multiprocessing.current_process().name
```

该语句会提供父进程的名字。

如何在后台运行进程

在后台运行进程是一种典型的耗时处理执行模式，它不需要你的介入与干预，当然也可以与其他程序并发执行。Python 的 multiprocessing 模块通过 daemon 选项可以实现进程的后台运行。

具体操作

为了运行后台进程，只需编写如下代码：

```
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    time.sleep(3)
    print ("Exiting %s \n" %name)


if __name__ == '__main__':
    background_process = multiprocessing.Process\
                         (name='background_process',\
                          target=foo)
    background_process.daemon = True
    NO_background_process = multiprocessing.Process\
                            (name='NO_background_process',\
                             target=foo)

    NO_background_process.daemon = False

    background_process.start()
    NO_background_process.start()
```

为了在命令提示符中运行该脚本，请输入如下命令：

```
python background_process.py
```

该命令最终的输出如下所示：

```
C:\Python CookBook\ Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python background_process.py
```

```
Starting NO_background_process
Exiting NO_background_process
```

实例精解

为了在后台执行进程，我们设置了 daemon 参数：

```
background_process.daemon = True
```

非后台模式下的进程有输出，因此在主程序结束后，后台进程会自动结束，从而避免进程的持续运行。

知识扩展

注意，后台进程是不允许创建子进程的。否则，当后台进程的父进程退出时，它会终止，这会导致这个后台进程的子进程变成游离状态。此外，它们并非 UNIX 守护进程或服务，它们就是正常的进程，如果非后台进程退出时，它们就会终止。

如何杀死进程

可以通过 terminate() 方法立刻杀死一个进程。此外，还可以通过 is_alive() 方法来追踪进程是否存活。

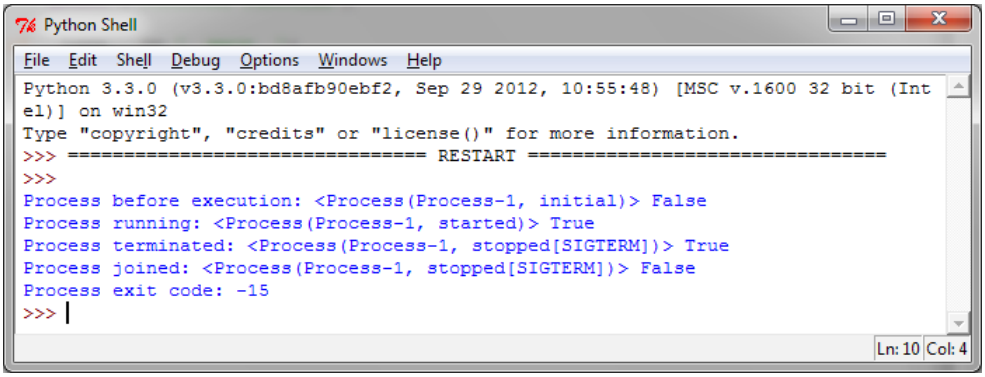
具体操作

在该示例中，进程是通过目标函数 foo() 来创建的。进程启动后，我们通过 terminate() 函数将其杀死：

```
# 杀死进程：第 3 章：基于进程的并行
import multiprocessing
import time

def foo():
    print ('Starting function')
    time.sleep(0.1)
    print ('Finished function')

if __name__ == '__main__':
    p = multiprocessing.Process(target=foo)
    print ('Process before execution:', p, p.is_alive())
    p.start()
    print ('Process running:', p, p.is_alive())
```

```
p.terminate()
print ('Process terminated:', p, p.is_alive())
p.join()
print ('Process joined:', p, p.is_alive())
print ('Process exit code:', p.exitcode)
```

下图所示的是执行上述命令后的输出结果。



实例精解

我们创建了进程，然后通过 is_alive() 方法监控其生命周期。接下来，调用 terminate() 方法终止进程：

```
p.terminate()
```

最后，当进程终止时我们验证了状态码，并读取了 ExitCode 属性值。ExitCode 的可能值有如下几项。

- == 0：表示没有错误。
- > 0：表示进程遇到了错误并退出。
- < 0：表示进程被信号 *-1 * ExitCode* 杀死了。

对于该示例来说，ExitCode 代码的输出值为 -15。负值 -15 表示子进程被数字 15 所标识的中断信号终止了。

如何在子类中使用进程

要想实现自定义子类与进程，需要：

- 定义一个新的 Process 类的子类。
- 重写 __init__(self [,args]) 方法增加额外的参数。
- 重写 run(self [,args]) 方法实现 Process 启动后需要做的事情。

创建好新的 Process 子类后，你可以创建它的一个实例，然后通过调用 start() 方法来启动，该方法又会调用 run() 方法。

具体操作

我们来以下面的方式重写第一个示例：

```
#在子类中使用进程，第3章：基于进程的并行

import multiprocessing

class MyProcess(multiprocessing.Process):
    def run(self):
        print ('called run method in process: %s' %self.name)
        return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = MyProcess ()
        jobs.append(p)
        p.start()
        p.join()
```

要想在命令提示符中运行该脚本，请输入如下命令：

```
python subclass_process.py
```

上述命令的输出结果如下所示：

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python subclass_process.py

called run method in process: MyProcess-1
called run method in process: MyProcess-2
called run method in process: MyProcess-3
called run method in process: MyProcess-4
called run method in process: MyProcess-5
```

实例精解

每个 Process 子类都可以由一个继承自 Process 类且重写了其 run() 方法的类来表示。该方法是 Process 的起始点：

```
class MyProcess (multiprocessing.Process):
    def run(self):
```

```
        print ('called run method in process: %s' %self.name)
        return
```

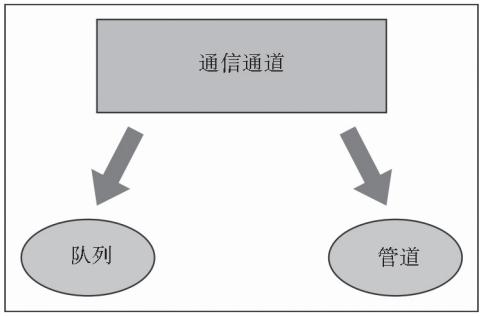
在主程序中，我们创建了 MyProcess() 类型的几个对象。当 start() 方法被调用后，线程就会开始执行：

```
p = MyProcess()
p.start()
```

join() 命令会处理进程的终止。

如何在进程间交换对象

并行应用的开发需要在进程间进行数据交换。multiprocessing 库有两个通信通道，通过它们可以管理对象的交换，分别是队列与管道，如下图所示。



multiprocessing 模块中的通信通道

使用队列进行对象交换

如前所述，我们可以通过队列数据结构来共享数据。

队列会返回一个进程共享队列，它是线程与进程安全的，任何可序列化对象（Python 使用 pickable 模块来序列化对象）都可以通过它进行交换。

具体操作

如下示例展示了如何使用队列来解决生产者 - 消费者问题。producer 类创建了条目与队列，consumer 类则提供了移除所插入的条目的方法：

```
import multiprocessing
import random
import time

class producer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print ("Process Producer : item %d appended to queue %s"\
                   % (item,self.name))
            time.sleep(1)
            print ("The size of queue is %s"\
                   % self.queue.qsize())

class consumer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            if (self.queue.empty()):
                print("the queue is empty")
                break
            else :
                time.sleep(2)
                item = self.queue.get()
                print ('Process Consumer : item %d popped from by %s \n'\
                       % (item, self.name))
                time.sleep(1)
    if __name__ == '__main__':
            queue = multiprocessing.Queue()
            process_producer = producer(queue)
            process_consumer = consumer(queue)
            process_producer.start()
            process_consumer.start()
            process_producer.join()
            process_consumer.join()
```

下面是程序运行后的输出：

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python using_queue.py

Process Producer : item 69 appended to queue producer-1
The size of queue is 1
Process Producer : item 168 appended to queue producer-1
The size of queue is 2
Process Consumer : item 69 popped from by consumer-2
Process Producer : item 235 appended to queue producer-1
The size of queue is 2
Process Producer : item 152 appended to queue producer-1
The size of queue is 3
Process Producer : item 213 appended to queue producer-1
Process Consumer : item 168 popped from by consumer-2
The size of queue is 3
Process Producer : item 35 appended to queue producer-1
The size of queue is 4
Process Producer : item 218 appended to queue producer-1
The size of queue is 5
Process Producer : item 175 appended to queue producer-1
Process Consumer : item 235 popped from by consumer-2
The size of queue is 5
Process Producer : item 140 appended to queue producer-1
The size of queue is 6
Process Producer : item 241 appended to queue producer-1
The size of queue is 7
Process Consumer : item 152 popped from by consumer-2
Process Consumer : item 213 popped from by consumer-2
Process Consumer : item 35 popped from by consumer-2
Process Consumer : item 218 popped from by consumer-2
Process Consumer : item 175 popped from by consumer-2
Process Consumer : item 140 popped from by consumer-2
Process Consumer : item 241 popped from by consumer-2
the queue is empty
```

实例精解

multiprocessing 类在主程序中实例化了 Queue 对象：

```
if __name__ == '__main__':
       queue = multiprocessing.Queue()
```

接下来创建了两个进程，producer 与 consumer，并将 Queue 对象作为一个属性：

```
process_producer = producer(queue)
process_consumer = consumer(queue)
```

进程 producer 负责通过其 put() 方法向队列中插入 10 个条目：

```
for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
```

进程 consumer 的任务是从队列中移除条目（使用 get 方法），并验证队列不为空。如果队列为空，那么 while 循环中的流程就会经由 break 语句而终止：

```
def run(self):
        while True:
            if (self.queue.empty()):
                print("the queue is empty")
                break
            else :
                time.sleep(2)
                item = self.queue.get()
                print ('Process Consumer : item %d popped from by %s
\n'\
                       % (item, self.name))
                time.sleep(1)
```

知识扩展

队列中有一个 JoinaleQueue 子类，它拥有如下两个附加的方法。

- task_done()：这表示一个任务已经执行完毕，比如，在 get() 方法从队列中获取条目之后。因此，它只能被队列消费者所使用。
- join()：这会阻塞进程，直到队列中的所有条目都被获取并处理完毕。

使用管道进行对象交换

第二种交换通道是管道数据结构。

管道会完成如下事情：

- 返回由管道所连接的一对连接对象。
- 在这里，每个对象都拥有 send/receive 方法，实现进程间通信。

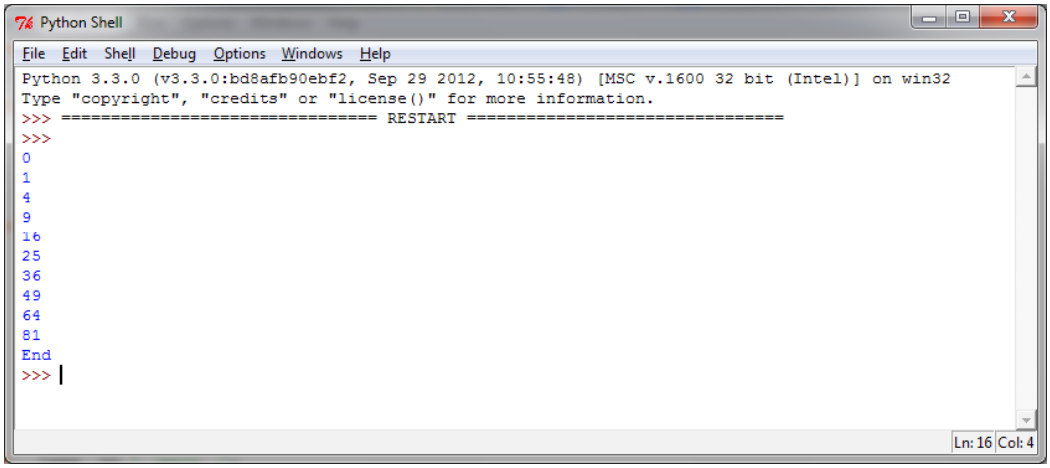
具体操作

下面是管道的一个简单示例。我们有一个进程管道，它会生成 0 到 9 这 10 个数字，另一个进程会接收到这些数字并将其进行乘方：

```
import multiprocessing


def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()

def multiply_items(pipe_1, pipe_2):
    close, input_pipe = pipe_1
    close.close()
    output_pipe, _ = pipe_2
    try:
        while True:
            item = input_pipe.recv()
            output_pipe.send(item * item)
    except EOFError:
        output_pipe.close()

if __name__== '__main__':

#拥有数字0~9的第一个进程管道
    pipe_1 = multiprocessing.Pipe(True)
    process_pipe_1 = \
                   multiprocessing.Process\
                   (target=create_items, args=(pipe_1,))
    process_pipe_1.start()

#第二个管道
    pipe_2 = multiprocessing.Pipe(True)
    process_pipe_2 = \
                   multiprocessing.Process\
```

```
                (target=multiply_items, args=(pipe_1, pipe_2,))
    process_pipe_2.start()

    pipe_1[0].close()
    pipe_2[0].close()

    try:
        while True:

            print (pipe_2[1].recv())
    except EOFError:
        print("End")
```

输出如下图所示。



实例精解

回忆一下，pipe() 函数会返回由一个双向管道所连接的一对连接对象。在该示例中，out_pipe 包含了数字 0 到 9，它是由目标函数 create_items() 生成的：

```
def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()
```

在第二个进程中有两个管道：输入管道与包含了结果的输出管道：

```
process_pipe_2 = multiprocessing.Process(target=multiply_items,
                     args=(pipe_1, pipe_2,))
```

下面是打印语句：

```
try:
        while True:
            print (pipe_2[1].recv())
except EOFError:
        print ("End")
```

如何同步进程

多个进程可以协同工作来执行一个给定的任务。通常情况下，它们会共享数据。多个进程对于共享数据的访问不会产生不一致的数据是很重要的。因此，通过共享数据进行协作的进程必须要按照一定的顺序来访问数据。同步原语类似于之前介绍的库与线程所用的那些同步原语。

它们如下所示。

- **Lock**：该对象可以处于上锁与未上锁状态。锁对象有两个方法：acquire() 与 release()，用于管理对共享资源的访问。
- **事件**：它实现了进程间的简单通信，一个进程会发出事件，其他进程会等待事件。Event 对象有两个方法：set() 与 clear()，用于管理内部的标志。
- **条件**：该对象用于同步串行或是并行进程中的部分工作流。它有两个基本的方法，wait() 用于等待条件，而 notify_all() 则用于与所应用的条件进行通信。
- **信号量**：用于共享公共资源，比如，支持固定数量的同时连接。
- **RLock**：定义了递归的 lock 对象。RLock 的方法和功能与 Threading 模块中的一样。
- **屏障**：将一个程序划分为几个阶段，因为它要求所有进程都到达后才能开始执行。屏障后的代码不能与屏障前的代码并发执行。

具体操作

该示例展示了如何使用 barrier() 来同步两个进程。我们有 4 个进程，其中进程 1 与进程 2 由屏障语句所管理，进程 3 与进程 4 则没有使用同步指令：

```
import multiprocessing
from multiprocessing import Barrier, Lock, Process
from time import time
from datetime import datetime

def test_with_barrier(synchronizer, serializer):
```

```
    name = multiprocessing.current_process().name
    synchronizer.wait()
    now = time()
    with serializer:
        print("process %s ----> %s" \
              %(name,datetime.fromtimestamp(now)))

def test_without_barrier():
    name = multiprocessing.current_process().name
    now = time()
    print("process %s ----> %s" \
          %(name ,datetime.fromtimestamp(now)))

if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - test_with_barrier'\
            ,target=test_with_barrier,\
            args=(synchronizer,serializer)).start()
    Process(name='p2 - test_with_barrier'\
            ,target=test_with_barrier,\
            args=(synchronizer,serializer)).start()
    Process(name='p3 - test_without_barrier'\
        ,target=test_without_barrier).start()
Process(name='p4 - test_without_barrier'\
        ,target=test_without_barrier).start()
```

脚本运行后，我们会看到进程 1 与进程 2 打印出了相同的时间戳：

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python process_barrier.py

process p1 - test_with_barrier ----> 2015-05-09 11:11:33.291229
process p2 - test_with_barrier ----> 2015-05-09 11:11:33.291229
process p3 - test_without_barrier ----> 2015-05-09 11:11:33.310230
process p4 - test_without_barrier ----> 2015-05-09 11:11:33.333231
```

实例精解

在主程序中，我们创建了 4 个进程；不过，还需要一个屏障和锁原语。屏障语句中的参数 2 表示将要管理的进程数量：

```
if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - test_with_barrier'\
            ,target=test_with_barrier,\
            args=(synchronizer,serializer)).start()
    Process(name='p2 - test_with_barrier'\
            ,target=test_with_barrier,\
            args=(synchronizer,serializer)).start()
```

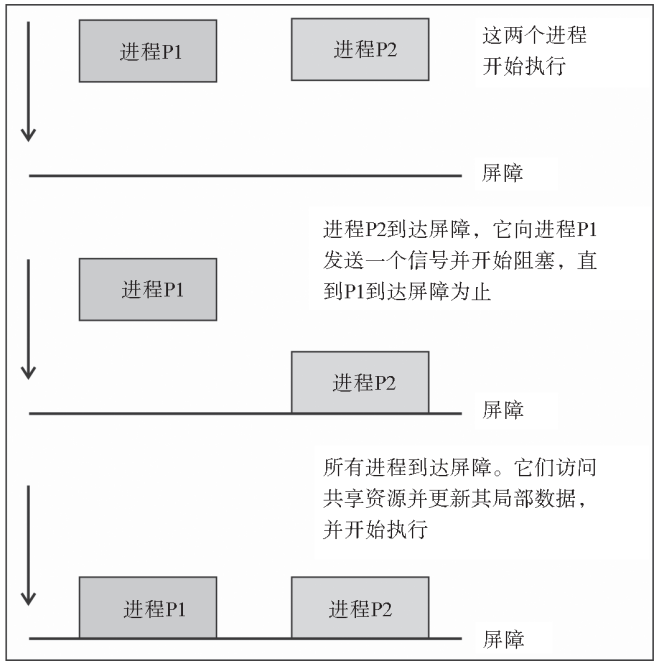
test_with_barrier_function 会执行屏障的 wait() 方法：

```
def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    synchronizer.wait()
```

当两个进程都调用了 wait() 方法后，它们就会同时被释放：

```
now = time()
with serializer:
    print("process %s ----> %s" %(name \
          ,datetime.fromtimestamp(now)))
```

下图展示了一个屏障是如何处理两个进程的。



使用屏障来管理进程

如何管理进程间状态

Python 的 multiprocessing 模块提供了一种管理器来协调用户之间的共享信息。管理器对象会控制一个服务端进程，该进程持有 Python 对象，并可以让其他进程操作这些对象。

管理器有如下属性：

- 它会控制服务端进程，该进程会管理共享对象。
- 当有人修改共享对象时，它会确保共享对象在所有进程中都会更新。

具体操作

下面通过一个示例来看看如何在进程间共享状态：

1. 首先，程序会创建一个管理器列表，并在 *n* 个 `taskWorker` 中共享它，每个执行者都会更新索引。
2. 当所有执行者执行完毕后，新的列表会被打印到标准输出：

```
import multiprocessing

def worker(dictionary, key, item):
    dictionary[key] = item

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    dictionary = mgr.dict()
    jobs = [ multiprocessing.Process\
             (target=worker, args=(dictionary, i, i*2))
             for i in range(10)
             ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print ('Results:', dictionary)
```

输出如下所示：

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python manager.py
key = 0 value = 0
key = 2 value = 4
key = 6 value = 12
```

```
key = 4 value = 8
key = 8 value = 16
key = 7 value = 14
key = 3 value = 6
key = 1 value = 2
key = 5 value = 10
key = 9 value = 18
Results: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9:
18}
```

实例精解

我们通过如下语句声明了管理器：

```
mgr = multiprocessing.Manager()
```

在接下来的语句中，创建了一个字典类型的数据结构：

```
dictionary = mgr.dict()
```

接下来加载 multiprocess：

```
jobs = [multiprocessing.Process \
        (target=taskWorker,args=(dictionary,i,i*2))
        for i in range(10)
       ]

for j in jobs:
        j.start()
```

这里，目标函数 taskWorker 会向数据结构字典中添加一个条目：

```
def taskWorker(dictionary, key, item):
    dictionary[key] = value
```

最后，获取到输出并将所有字典内容打印出来：

```
for j in jobs:
        j.join()
    print ('Results:', d)
```

如何使用进程池

multiprocessing 库提供了 Pool 类用于实现简单的并行处理任务。Pool 类拥有如下方法。

- apply()：一直会阻塞，直到结果就绪为止。
- apply_async()：这是 apply() 方法的一个变种，会返回一个结果对象。它是一个异

步操作，并不会锁定主线程，直到所有子类都执行完毕为止。

- map()：这是内建的 map() 函数的并行版本。它会阻塞住，直到结果就绪为止，该方法会将迭代的数据以块的形式提交给进程池，并作为单独的任务来执行。
- map_async()：这是 map() 方法的一个变种，会返回一个结果对象。如果指定了回调，那么它就是可以调用的，并且会接收一个参数。当结果就绪时，回调就会使用到它（除非调用失败了）。回调应该立即完成；否则，处理结果的线程就会被阻塞住。

具体操作

该示例介绍了如何实现一个进程池来执行一个并行应用。我们创建了一个由 4 个进程组成的进程池，然后使用进程池的 map 方法来执行一个简单的计算：

```
def function_square(data):
    result = data*data
    return result


if __name__ == '__main__':
    inputs = list(range(100))
    pool = multiprocessing.Pool(processes=4)
    pool_outputs = pool.map(function_square, inputs)
    pool.close()
    pool.join()
    print ('Pool    :', pool_outputs)
```

下面是计算完毕后的结果：

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>\python process_pool.py

Pool : [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784,
841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704,
2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096,
4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776,
5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744,
7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]
```

实例精解

multiprocessing.Pool 方法会将 function_square 应用到输入元素上来执行一个简单的计算。并行进程的总数量是 4：

```
pool = multiprocessing.Pool(processes=4)
```

pool.map 方法会将它们以单独的任务形式提交给进程池：

```
pool_outputs = pool.map(function_square, inputs)
```

参数 inputs 是一个从 0 到 100 的整数列表：

```
inputs = list(range(100))
```

计算的结果会被存储到 pool_outputs 中。接下来，将最终的结果打印出来：

```
print ('Pool        :' , pool_outputs)
```

值得注意的是，pool.map() 方法的结果等于 Python 内建函数 map() 的结果，区别在于进程是并行执行的。

使用mpi4py模块

Python 编程语言提供了大量的 MPI 模块来编写并行程序，其中最有趣的一个就是 mpi4py 库。它构建在 MPI-1/2 规范之上，并提供了一个面向对象的接口，仅仅跟随 MPI-2 C++ 绑定。如果你是一个 C MPI 用户，那么使用该模块时就无须学习新的接口了。因此，它在 Python 中几乎可以当作一个全功能型的 MPI 库来用。

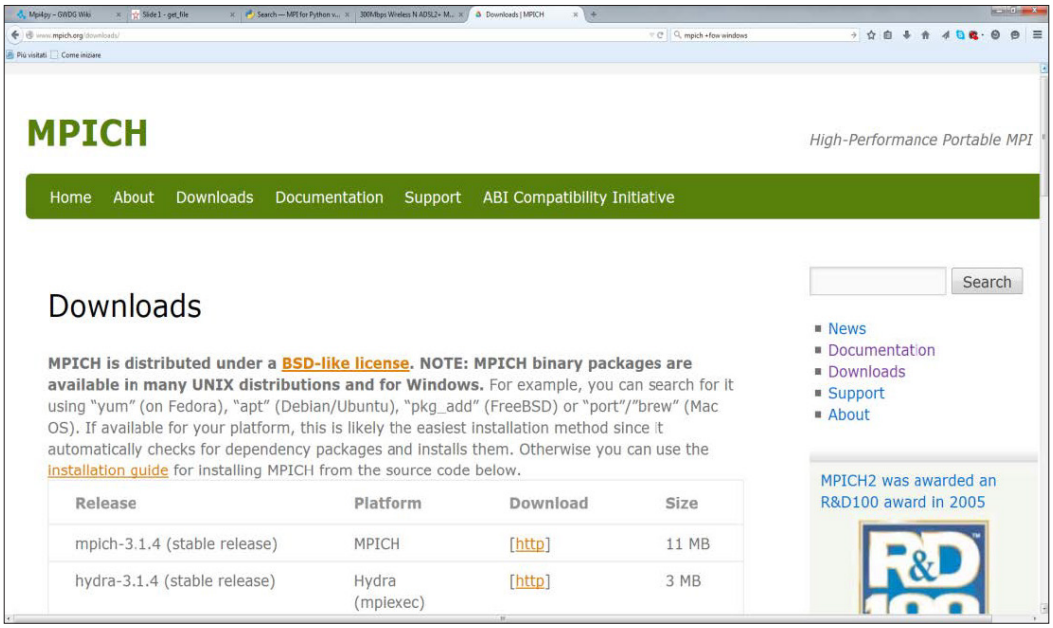
本章后续将会介绍的该模块的主要应用如下所示：

- 点对点通信
- 聚合通信
- 拓扑

准备工作

下面介绍在 Windows 中安装 mpi4py 的过程（对于其他操作系统，请参考 http://mpi4py.scipy.org/docs/usrman/install.html#）：

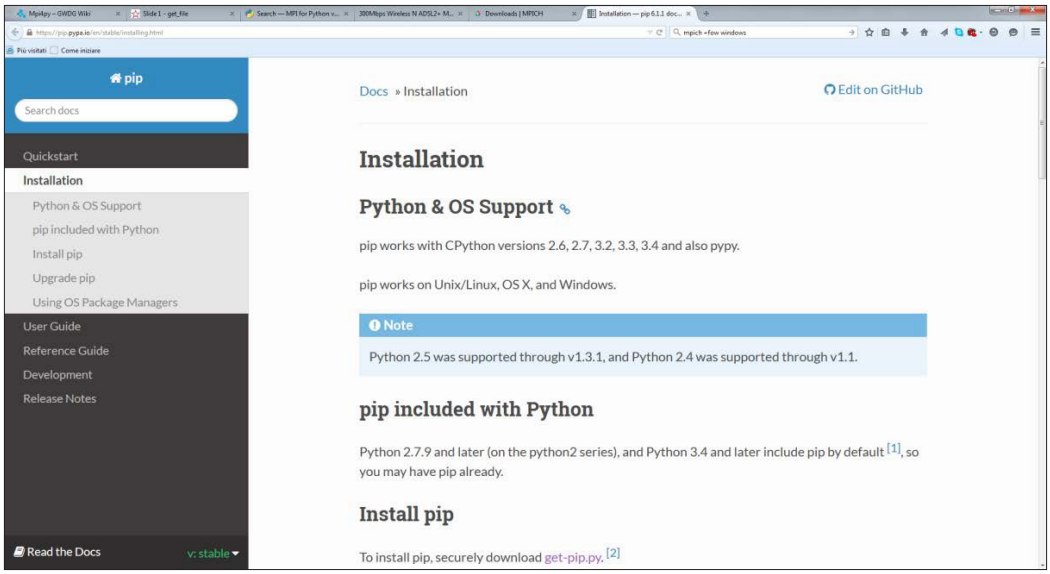
1. 从 http://www.mpich.org/downloads/ 下载 MPI 软件库 mpich，下载页面如下图所示。



MPICH 下载页面

2. 右键单击命令提示符图标，选择以管理员身份运行来打开管理员命令提示符。
3. 从管理员命令提示符中运行 `msiexec /i mpich_installation_file.msi` 来安装 MPICH2。
4. 在安装过程中，选择为所有用户安装 MPICH2 这一选项。
5. 运行 `wmpiconfig` 并存储用户名与密码。使用真实的 Windows 登录用户名与密码。
6. 将 `C:\Program Files\MPICH2\bin` 添加到系统路径中，无须重启机器。
7. 使用 `smpd -status` 来检查 `smpd`，应该返回 `smpd running on $hostname$`。
8. 要想测试执行环境，请进入 `$MPICHROOT\examples` 目录并使用 `mpiexec -n 4 cpi` 来运行 `cpi.exe`。
9. 从 `https://pip.pypa.io/en/stable/installing.html` 下载 Python 安装器 `pip`，如下图所示。

   这会在 Python 安装目录的 `Scripts` 目录中创建一个 `pip.exe` 文件。



PIP 下载页面

10. 接下来在命令提示符中输入如下命令来安装 mpi4py：

```
C:> pip install mpi4py
```

具体操作

我们通过在每一个实例化的进程中打印出“Hello, world!”这一经典程序来开启 MPI 库的探索之旅：

```
#hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print ("hello world from process ", rank)
```

为了执行代码，请输入如下命令：

```
C:> mpiexec -n 5 python helloWorld_MPI.py
```

执行结果如下所示：

```
('hello world from process ', 1)
('hello world from process ', 0)
('hello world from process ', 2)
('hello world from process ', 3)
('hello world from process ', 4)
```

实例精解

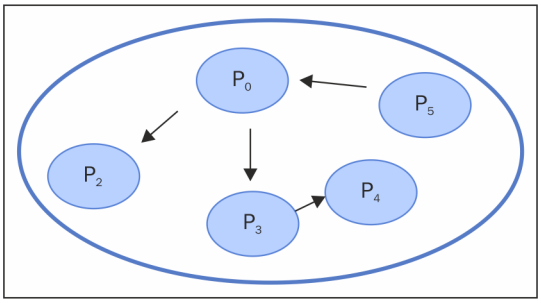
在 MPI 中，并行程序执行过程中所涉及的进程可以由一个非负的整数序列进行标识，这些整数叫作等级（rank）。如果一个程序有 *p* 个进程在运行，那么进程的等级就会从 0 到 *p*-1。用于解决问题的 MPI 提供了如下函数调用：

```
rank = comm.Get_rank()
```

该函数会返回调用它的那个进程的等级。comm 参数叫作通信器，因为它定义了自己的一套可互相通信的进程，即：

```
comm = MPI.COMM_WORLD
```

进程间的通信示例如下图所示。



MPI.COMM_WORLD 中进程间的通信示例

知识扩展

值得注意的是，出于说明的目的，标准输出上的输出并非总是有序的，因为多个进程会同时向屏幕写内容，操作系统则会随意选择顺序。因此，我们得到了这样一个基本的结论：MPI 执行过程中所涉及的每个进程都会运行相同的编译好的二进制代码，这样每个进程都会收到同样的待执行指令。

点对点通信

MPI 所提供的重要特性之一就是点对点通信，这指的是可以在两个进程间传递数据：一个进程接收者，一个进程发送者。

Python 模块 mpi4py 通过以下两个函数来实现点对点通信。

- `Comm.Send(data, process_destination)`：将数据发送给目标进程，目标进程是由其在通信器组中的等级来标识的。

- Comm.Recv(process_source)：从源进程接收数据，源进程也是由其在通信器组中的等级来标识的。

Comm 参数表示通信器，它定义了进程组，可以通过消息传递来通信：

```
comm = MPI.COMM_WORLD
```

具体操作

如下示例展示了如何使用 comm.send 与 comm.recv 指令在不同进程间交换消息：

```
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)

if rank==0:
    data= 10000000
    destination_process = 4
    comm.send(data,dest=destination_process)
    print ("sending data %s " %data + \
           "to process %d" %destination_process)

if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
    print ("sending data %s :" %data + \
           "to process %d" %destination_process)

if rank==4:
    data=comm.recv(source=0)
    print ("data received is = %s" %data)


if rank==8:
    data1=comm.recv(source=1)
    print ("data1 received is = %s" %data1)
```

输入如下命令来运行该脚本：

```
C:\>mpiexec -n 9 python pointToPointCommunication.py
```

下面是运行脚本后得到的输出：

```
('my rank is : ', 5)
('my rank is : ', 1)
sending data hello :to process 8
('my rank is : ', 3)
('my rank is : ', 0)
sending data 10000000 to process 4
('my rank is : ', 2)
('my rank is : ', 7)
('my rank is : ', 4)
data received is = 10000000
('my rank is : ', 8)
data1 received is = hello
('my rank is : ', 6)
```

实例精解

我们使用 9 个进程来运行该示例，这样在通信器组 comm 中，就有 9 个可以彼此通信的任务：

```
comm=MPI.COMM_WORLD
```

此外，为了标识出组中的任务与进程，我们使用了它们的 rank 值：

```
rank = comm.rank
```

我们有两个发送者进程和两个接收者进程。

rank 值为 0 的进程会向 rank 值为 4 的接收者进程发送一个数字数据：

```
if rank==0:
    data= 10000000
    destination_process = 4
    comm.send(data,dest=destination_process)
```

与之类似，我们必须要指定 rank 值为 4 的接收者进程。此外，值得注意的是，comm.recv 语句必须包含一个参数，该参数指定了发送者进程的 rank 值：

```
…
if rank==4:
    data=comm.recv(source=0)
```

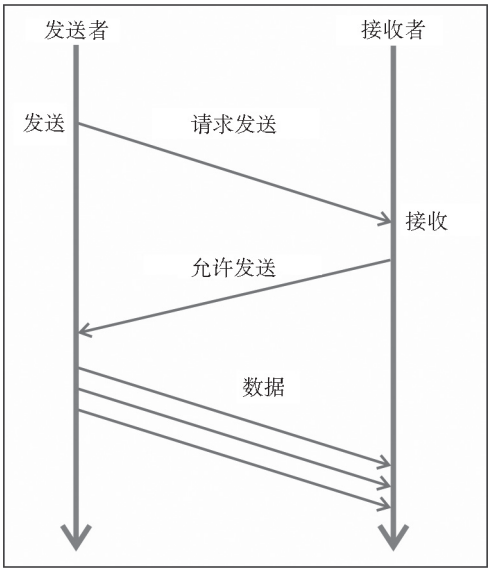
对于另一对发送者与接收者进程来说，它们的 rank 值分别为 1 和 8，情况一样，唯一的差别在于数据类型的不同。在该情况中，我们发送的是一个字符串：

```
if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
```

对于 rank 值为 8 的接收者进程来说，其发送者进程的 rank 值是这样指定的：

```
if rank==8:
    data1=comm.recv(source=1)
```

下图总结了在 mpi4py 中实现的点对点通信协议：



发送 / 接收传输协议

这是一个两步骤的过程，包括从一个任务（发送者）发送数据和被另一个任务（接收者）接收这些数据。发送任务必须要指定待发送的数据及目的地（接收者进程），接收任务则要指定所要接收的消息的来源。

知识扩展

comm.send() 与 comm.recv() 函数是阻塞函数；它们会阻塞调用者，直到缓冲数据被安全地使用为止。在 MPI 中，有两种发送与接收消息的方法：

- 缓冲模式
- 同步模式

在缓冲模式中，当待发送的数据被复制到缓冲区后，流程控制就会返回到程序中。这并不意味着消息已经被发送或是接收了。不过在同步模式中，只有在相应的接收函数开始接收消息时，函数才会终止。

避免死锁问题

我们所面临的一个常见问题就是死锁。死锁指的是两个或多个进程彼此阻塞，一个进程等待另一个进程执行某个动作来满足自己的需要，反之亦然。mpi4py 模块并未提供任何具体的功能来解决这一问题，它只是提供了一些举措，开发者需要遵循这些举措来避免死锁问题。

具体操作

我们先来分析如下 Python 代码，这段代码会引入一个典型的死锁问题；我们有两个进程，其 rank 值分别等于 1 和 5，它们之间彼此通信，各自都拥有数据发送者与数据接收者的功能：

```
from mpi4py import MPI

comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)

if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5

    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)

    print ("sending data %s " %data_send + \
           "to process %d" %destination_process)
    print ("data received is = %s" %data_received)



if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1

    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)


    print ("sending data %s :" %data_send + \
           "to process %d" %destination_process)
 print ("data received is = %s" %data_received)
```

实例精解

如果运行该程序（只使用两个进程来运行就可以），会发现这两个进程都无法继续下去：

```
C:\>mpiexec -n 9 python deadLockProblems.py
('my rank is : ', 8)
('my rank is : ', 3)
('my rank is : ', 2)
('my rank is : ', 7)
('my rank is : ', 0)
('my rank is : ', 4)
('my rank is : ', 6)
```

这两个进程都准备从对方那里接收消息，但都卡住了。这是因为函数 comm.recv()MPI 与 comm.send() MPI 会将其阻塞住。这意味着调用进程会等待其完成。对于 comm.send() MPI 来说，完成指的是数据已经被发送出去，并且可以在不修改消息的情况下被覆盖。与之相反，comm.recv() MPI 的完成指的是数据已经被接收到且可以使用。为了解决这一问题，我们首先想到的是交换 comm.recv() MPI 与 comm.send() MPI 的位置，如下所示：

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)
```

不过，这个解决方案虽然从逻辑的视角来看是正确的，但它无法总能避免死锁问题。由于通信是经由缓冲区进行的，而缓冲区是 comm.send() MPI 复制待发送数据的地方，因此程序能够平滑运行的前提是该缓冲区可以承载所有数据，否则就会导致死锁：发送者无法发送完数据，因为缓冲区已满，而接收者无法接收到数据，因为它被 comm.send() MPI 阻塞了，无法完成。这时，避免死锁的一种解决方案就是交换发送与接收函数的位置，使得它们变成非对称的：

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send,dest=destination_process)
```

```
    data_received=comm.recv(source=source_process)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)
```

最后，我们得到了正确的输出：

```
C:\>mpiexec -n 9 python deadLockProblems.py

('my rank is : ', 7)
('my rank is : ', 0)
('my rank is : ', 8)
('my rank is : ', 1)
sending data a to process 5
data received is = b
('my rank is : ', 5)
sending data b :to process 1
data received is = a
('my rank is : ', 2)
('my rank is : ', 3)
('my rank is : ', 4)
('my rank is : ', 6)
```

知识扩展

上面并非死锁问题的唯一解决方案。比如，有这样一个特殊函数，它统一了向给定进程发送消息的调用与接收来自另外一个线程的消息的调用。该函数叫作 Sendrecv：

```
Sendrecv(self, sendbuf, int dest=0, int sendtag=0, recvbuf=None, int
source=0, int recvtag=0, Status status=None)
```

如你所见，所需的参数与 comm.send()MPI 和 comm.recv() MPI 相同。此外，在该示例中，函数会阻塞，不过相比于之前看到的两个函数来说，它的优势在于让通信子系统负责检查发送与接收的依赖关系，从而避免了死锁。通过这种方式，上述示例的代码变成了下面这样：

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    data_received=comm.sendrecv(data_send,dest=destination_process,
                                source =source_process)
if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.sendrecv(data_send,dest=destination_process,
                                source=source_process)
```

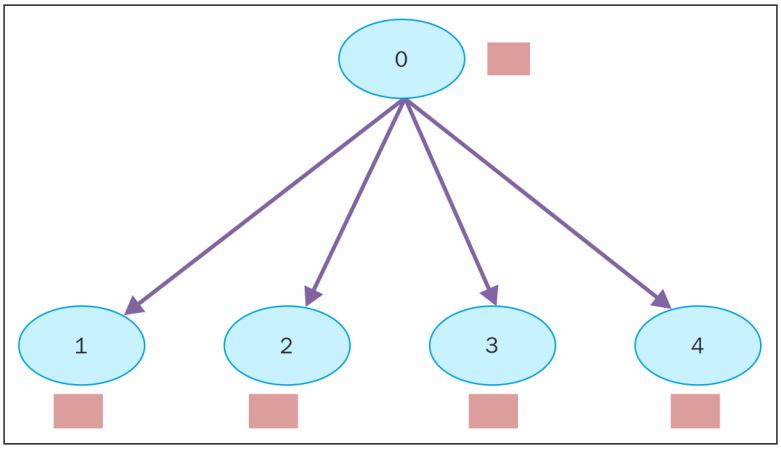
使用广播实现聚合通信

在并行代码的开发过程中，我们常常会遇到这样的情况，即运行期需要在多个进程间共享某个变量的值，或是共享由每个进程所提供的对变量的操作（操作不同的值）。

为了解决这类问题，我们使用了通信树（比如，进程 0 向进程 1 与 2 发送数据，进程 1 与 2 则分别将数据发送给进程 3、4、5、6 等）。

MPI 库提供了一些非常适合在多个进程间进行信息交换或是使用的函数，它们在所执行的机器上进行了很好的优化。

从一个进程向其他进程进行数据广播的示意图如下图所示。



从进程 0 到进程 1、2、3、4 的数据广播

涉及属于某个通信器的所有进程的通信方法叫作聚合通信。因此，聚合通信一般来说会涉

及两个以上的进程。不过，我们会将聚合通信叫作广播，其中一个进程向其他进程发送相同的数据。广播中的 mpi4py 功能是由如下函数提供的：

```
buf = comm.bcast(data_to_share, rank_of_root_process)
```

该函数只是将消息进程根中的信息发送给属于 comm 通信器的其他进程；不过，每个进程都必须要通过相同的 root 与 comm 值来调用它。

具体操作

接下来的示例使用了广播函数。我们有一个根进程，其 rank 值等于 0，它与定义在通信器组中的其他进程共享数据 variable_to_share：

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   variable_to_share = 100

else:
   variable_to_share = None

variable_to_share = comm.bcast(variable_to_share, root=0)
print("process = %d" %rank + " variable shared  = %d " \
                     %variable_to_share)
```

下面是包含了 10 个进程的通信器组的输出结果：

```
C:\>mpiexec -n 10 python broadcast.py
process = 0 variable shared = 100
process = 8 variable shared = 100
process = 2 variable shared = 100
process = 3 variable shared = 100
process = 4 variable shared = 100
process = 5 variable shared = 100
process = 9 variable shared = 100
process = 6 variable shared = 100
process = 1 variable shared = 100
process = 7 variable shared = 100
```

实例精解

rank 值为 0 的进程根将变量 variabile_to_share 的值实例化为 100。该变量会与通信器组中的其他进程共享：

```
if rank == 0:
   variable_to_share = 100
```

为了做到这一点，我们还加上了广播通信语句：

```
variable_to_share = comm.bcast(variable_to_share, root=0)
```

这里，函数中的参数分别是待共享的数据以及根进程或是主发送者进程，正如上面的图片中所展示的那样。当运行代码时，我们有一个拥有 10 个进程的通信组，variable_to_share 会在组中的其他进程中被共享。最后，print 语句会打印出运行中的进程的 rank 值及其变量的值：

```
print("process = %d" %rank + " variable shared = %d " \
                 %variable_to_share)
```

知识扩展

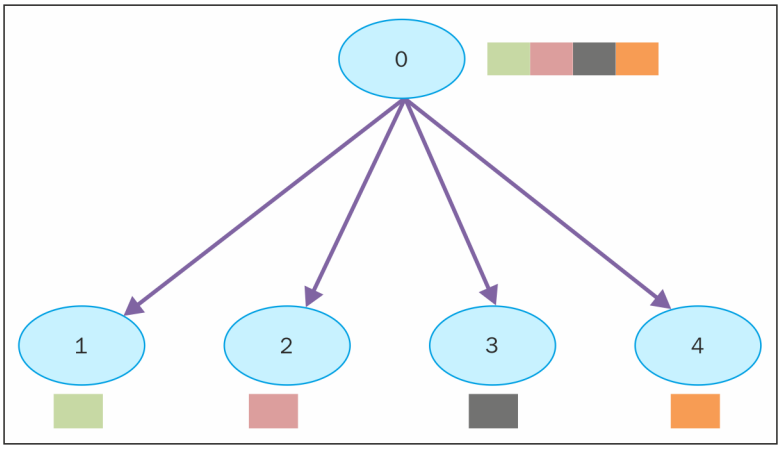
借助于聚合通信，我们可以实现一个组内的在多个进程间同时进行数据传递。mpi4py 只提供了聚合通信的阻塞版本（它会阻塞调用者方法，直到缓冲区数据可以安全使用为止）。

常见的聚合操作有如下几项。

- 跨越组内进程的屏障同步。
- 通信功能：
  - 从一个进程向组内的其他所有进程广播数据。
  - 将所有进程的数据收集到一个进程。
  - 从一个进程将数据分发到其他进程。
- 汇聚操作。

使用scatter实现聚合通信

scatter 的功能类似于 scatter 广播，不过有一个明显的区别，虽然 comm.bcast 会向所有监听进程发送同样的数据，但 comm.scatter 却能以数组形式将数据块发送给不同进程。下图展示了 scatter 的功能。



将数据从进程 0 散播给进程 1、2、3 与 4

comm.scatter 函数会接收数组元素，并根据进程的等级值将它们发送给相应的进程。第 1 个元素会被发送给进程 0，第 2 个元素会被发送给进程 1，以此类推。该函数在 mpi4py 中的实现如下所示：

```
recvbuf = comm.scatter(sendbuf, rank_of_root_process)
```

具体操作

在接下来的示例中，我们将会介绍如何通过 scatter 功能将数据分发给不同的进程：

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   array_to_share = [1, 2, 3, 4 ,5 ,6 ,7, 8 ,9 ,10]

else:
   array_to_share = None

recvbuf = comm.scatter(array_to_share, root=0)
print("process = %d" %rank + " recvbuf = %d " %array_to_share)
```

上述代码的输出结果如下所示：

```
C:\>mpiexec -n 10 python scatter.py
process = 0 variable shared = 1
```

```
process = 4 variable shared = 5
process = 6 variable shared = 7
process = 2 variable shared = 3
process = 5 variable shared = 6
process = 3 variable shared = 4
process = 7 variable shared = 8
process = 1 variable shared = 2
process = 8 variable shared = 9
process = 9 variable shared = 10
```

实例精解

rank 值为 0 的进程会将 array_to_share 数据结构分发给其他进程：

```
array_to_share = [1, 2, 3, 4 ,5 ,6 ,7, 8 ,9 ,10]
```

recvbuf 参数表示通过 comm.scatter 语句将要发送给第 *i* 个进程的第 *i* 个变量的值：

```
recvbuf = comm.scatter(array_to_share, root=0)
```

值得一提的是，comm.scatter 的一个限制是你可以散播与执行语句中所指定的进程数相同的元素个数。实际上，如果散播的元素数量超过了所指定的进程数（本例中为 3 个），那么会看到这样一个错误：

```
C:\> mpiexec -n 3 python scatter.py
Traceback (most recent call last):
  File "scatter.py", line 13, in <module>
    recvbuf = comm.scatter(array_to_share, root=0)
  File "Comm.pyx", line 874, in mpi4py.MPI.Comm.scatter (c:\users\utente\
appdata

\local\temp\pip-build-h14iaj\mpi4py\src\mpi4py.MPI.c:73400)
  File "pickled.pxi", line 658, in mpi4py.MPI.PyMPI_scatter (c:\users\
utente\app
data\local\temp\pip-build-h14iaj\mpi4py\src\mpi4py.MPI.c:34035)
  File "pickled.pxi", line 129, in mpi4py.MPI._p_Pickle.dumpv (c:\users\
utente\a
ppdata\local\temp\pip-build-h14iaj\mpi4py\src\mpi4py.MPI.c:28325)
ValueError: expecting 3 items, got 10
mpiexec aborting job...

job aborted:
rank: node: exit code[: error message]
```

```
0: Utente-PC: 123: mpiexec aborting job
1: Utente-PC: 123
2: Utente-PC: 123
```

知识扩展

mpi4py 库还提供了另外两个函数用于散播数据。

- comm.scatter(sendbuf, recvbuf, root=0)：会将相同通信器中一个进程的数据发送给所有其他进程。
- comm.scatterv(sendbuf, recvbuf, root=0)：会将一个组中一个进程的数据发送给所有其他进程。在发送端，它可以发送不同数量的数据，偏移量也可以不同。

sendbuf 与 recvbuf 参数必须要以列表的形式给出（类似于点对点函数 comm.send）：

```
buf = [data, data_size, data_type]
```

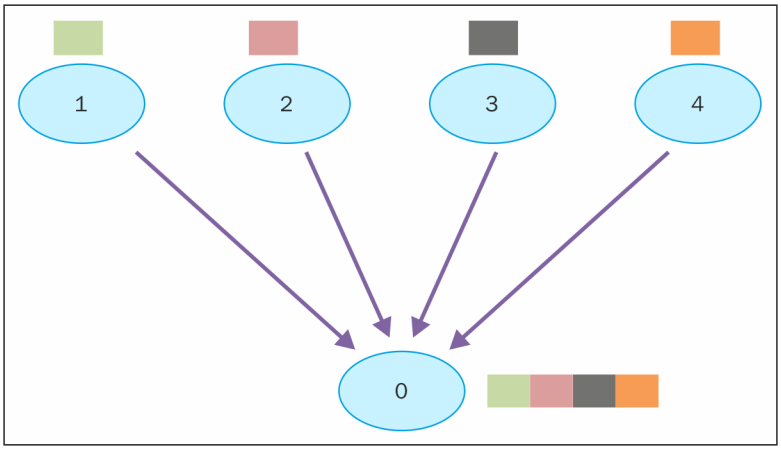
这里的 data 必须是一个类似于缓冲的对象，其大小为 data_size，类型为 data_type。

使用gather实现聚合通信

gather 函数会执行与 scatter 相反的功能。在该示例中，所有进程会向根进程发送数据，根进程则会收集所接收到的数据。gather 函数在 mpi4py 中的实现如下所示：

```
recvbuf = comm.gather(sendbuf, rank_of_root_process)
```

这里的 sendbuf 是发送的数据，rank_of_root_process 表示所有数据的进程接收者，示意图如下图所示。



从进程 1、2、3 与 4 收集数据

具体操作

在该示例中，我们想要表示上图中所描述的情况。每个进程会构建自己的数据，数据会被发送给 rank 值为 0 的根进程：

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
data = (rank+1)**2
data = comm.gather(data, root=0)
if rank == 0:
   print ("rank = %s " %rank +\
          "...receiving data to other process")
   for i in range(1,size):
      data[i] = (i+1)**2
      value = data[i]
      print(" process %s receiving %s from process %s"\
            %(rank , value , i))
```

最后，我们运行代码并将线程数设为 5：

```
C:\>mpiexec -n 5 python gather.py
rank = 0 ...receiving data to other process
 process 0 receiving 4 from process 1
 process 0 receiving 9 from process 2
 process 0 receiving 16 from process 3
 process 0 receiving 25 from process 4
```

根进程 0 会从其他 4 个进程接收数据，正如上图所展示的那样。

实例精解

我们有 *n* 个进程来发送数据：

```
data = (rank+1)**2
```

如果进程的 rank 值为 0，那么数据就会被收集到数组中：

```
if rank == 0:
   for i in range(1,size):
      data[i] = (i+1)**2
      value = data[i]
   ...
```

数据的收集是通过如下函数实现的：

```
data = comm.gather(data, root=0)
```

知道扩展

mpi4py 提供了如下函数来收集数据。

- **收集到一个任务**：comm.Gather、comm.Gatherv 与 comm.gather。
- **收集到所有任务**：comm.Allgather、comm.Allgatherv 与 comm.allgather。

使用Alltoall实现聚合通信

Alltoall 聚合通信整合了 scatter 与 gather 的功能。mpi4py 中有 3 类 Alltoall 聚合通信。

- comm.Alltoall(sendbuf, recvbuf)：all-to-all scatter/gather 从组中的 all-to-all 进程发送数据。
- comm.Alltoallv(sendbuf, recvbuf)：all-to-all scatter/gather 从组中的 all-to-all 进程发送数据，提供了不同的数据量与偏移量。
- comm.Alltoallw(sendbuf, recvbuf)：广义的 all-to-all 通信，支持每个进程使用不同数量、不同偏移量与不同数据类型的数据。

具体操作

在如下示例中，我们将会看到一个 comm.Alltoall 的 mpi4py 实现。我们考虑一个进程的通信器组，其中每个进程会向组中的其他进程发送一个数值类型的数组，也会从其他进程接收这样的数组。

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

a_size = 1
senddata = (rank+1)*numpy.arange(size,dtype=int)
recvdata = numpy.empty(size*a_size,dtype=int)
comm.Alltoall(senddata,recvdata)


print(" process %s sending %s receiving %s"\
      %(rank , senddata , recvdata))
```

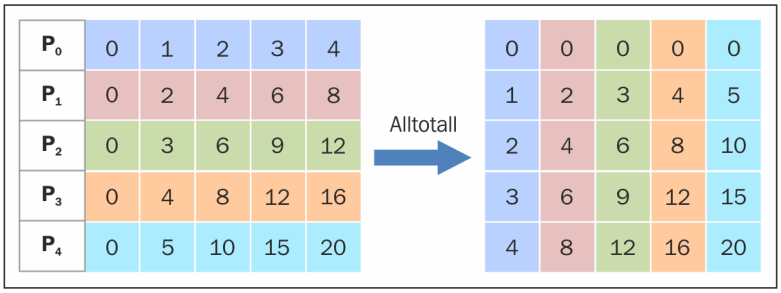
运行上述代码，将通信器组中的进程数指定为 5，输出结果如下所示：

```
C:\>mpiexec -n 5 python alltoall.py
process 0 sending [0 1 2 3 4] receiving [0 0 0 0 0]
process 1 sending [0 2 4 6 8] receiving [1 2 3 4 5]
process 2 sending [ 0 3 6 9 12] receiving [ 2 4 6 8 10]
process 3 sending [ 0 4 8 12 16] receiving [ 3 6 9 12 15]
process 4 sending [ 0 5 10 15 20] receiving [ 4 8 12 16 20]
```

实例精解

comm.Alltoall 方法从任务 *j* 的 sendbuf 中接收到第 *i* 个对象，然后将其复制到任务 *i* 的 recvbuf 参数的第 *j* 个对象处。

通过下面这张图可以更好地展示出所发生的事情：



Alltoall 聚合通信

接下来对这张图加以解释说明：

- 进程 P_0 包含数据数组 [0 1 2 3 4]，它将 0 赋给自身，将 1 赋给进程 P_1，将 2 赋给进程 P_2，将 3 赋给进程 P_3，将 4 赋给进程 P_4。
- 进程 P_1 包含数据数组 [0 2 4 6 8]，它将 0 赋给进程 P_0，将 2 赋给自身，将 4 赋给进程 P_2，将 6 赋给进程 P_3，将 8 赋给进程 P_4。
- 进程 P_2 包含数据数组 [0 3 6 9 12]，它将 0 赋给进程 P_0，将 3 赋给进程 P_1，将 6 赋给自身，将 9 赋给进程 P_3，将 12 赋给进程 P_4。
- 进程 P_3 包含数据数组 [0 4 8 12 16]，它将 0 赋给进程 P_0，将 4 赋给进程 P_1，将 8 赋给进程 P_2，将 12 赋给自身，将 16 赋给进程 P_4。
- 进程 P_4 包含数据数组 [0 5 10 15 20]，它将 0 赋给 P_0，将 5 赋给进程 P_1，将 10 赋给进程 P_2，将 15 赋给进程 P_3，将 20 赋给自身。

知识扩展

All-to-all 个性化通信又叫作全交换。该操作可用在各种并行算法中，如快速傅里叶变换、矩阵转置、抽样排序以及一些并行的数据库连接操作等。

汇聚操作

类似于 comm.gather，comm.reduce 也会在每个进程中接收一个输入元素的数组，并将一个输出元素的数组返回给根进程。输出元素包含了汇聚后的结果。

我们在 mpi4py 中通过如下语句定义了汇聚操作：

```
comm.Reduce(sendbuf, recvbuf, rank_of_root_process, op = type_of_
reduction_operation)
```

值得一提的是，comm.gather 语句现在位于 op 参数中了，它指的是你希望对数据所应用的操作，mpi4py 模块包含了一组可以使用的汇聚操作。由 MPI 所定义的一些汇聚操作有如下几项。

- MPI.MAX：返回最大的元素。
- MPI.MIN：返回最小的元素。
- MPI.SUM：对元素求和。
- MPI.PROD：将所有元素相乘。
- MPI.LAND：对元素执行一个逻辑运算。
- MPI.MAXLOC：返回最大值以及拥有该最大值的进程的 rank 值。
- MPI.MINLOC：返回最小值以及拥有该最小值的进程的 rank 值。

具体操作

现在，我们来看看如何通过汇聚功能 MPI.SUM 计算出一个数组中元素的和。每个进程会操作一个大小为 3 的数组。对于数组操作来说，我们使用了 Python 模块 numpy 所提供的函数：

```
import numpy
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.size
rank = comm.rank

array_size = 3
recvdata = numpy.zeros(array_size,dtype=numpy.int)
senddata = (rank+1)*numpy.arange(a_size,dtype=numpy.int)
print(" process %s sending %s " %(rank , senddata))
```

```
comm.Reduce(senddata,recvdata,root=0,op=MPI.SUM)
print ('on task',rank,'after Reduce: data = ',recvdata)
```

使用由 3 个进程所构成的通信器组来运行上述代码是合理的，即所操作的数组的大小。最后，结果如下所示：

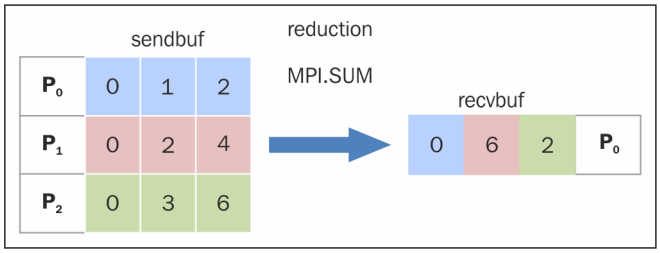
```
C:\>mpiexec -n 3 python reduction2.py
 process 2 sending [0 3 6]
on task 2 after Reduce: data = [0 0 0]
 process 1 sending [0 2 4]
on task 1 after Reduce: data = [0 0 0]
 process 0 sending [0 1 2]
on task 0 after Reduce: data = [ 0 6 12]
```

实例精解

为了执行汇聚操作求和，我们使用了 comm.Reduce 语句，并将根进程的 rank 值设为 0，它包含了 recvbuf，用于表示计算的最终结果：

```
comm.Reduce(senddata,recvdata,root=0,op=MPI.SUM)
```

此外，值得一提的是，借由 op=MPI.SUM 选项，我们将求和操作应用到了数组的所有元素上。为了更好地理解汇聚操作的执行过程，来看看下面这张图。



汇聚聚合通信

发送操作如下所示：

- 进程 P_0 发送数据数组 [0 1 2]
- 进程 P_1 发送数据数组 [0 2 4]
- 进程 P_2 发送数据数组 [0 3 6]

汇聚操作会求出每个任务第 *i* 个元素的和，然后将结果放到根进程 P_0 数组中的第 *i* 个元素的位置处。

对于接收操作来说，进程 P_0 会接收到数据数组 [0 6 12]。

如何优化通信

MPI 所提供的一个有趣的特性是关于虚拟拓扑的。如前所述，所有的通信功能（点对点或是聚合）都指的是一组进程。我们总是在使用 MPI_COMM_WORLD 组，它包含了所有进程。它会为属于大小为 *n* 的通信器的每一个进程分配一个从 0 到 *n*-1 的 rank 值。不过，我们可以通过 MPI 为通信器分配一个虚拟拓扑。它为不同的进程定义了特殊的标签。这种机制可以提升执行性能。实际上，如果构建了虚拟拓扑，那么每个节点都只会与其虚拟邻居通信，这优化了性能。

比如，如果 rank 值是随机分配的，那么消息在到达目的地前就会经过很多其他节点。除了性能问题以外，虚拟拓扑还可以确保代码更加清晰，可读性更好。MPI 提供了两种构建拓扑。第一种会创建笛卡儿拓扑，第二种则会创建其他类型的拓扑。特别的，对于第二种情况来说，我们必须要为想要构建的图提供邻接矩阵。这里只讨论笛卡儿拓扑，通过它可以构建出几种广泛使用的结构：如网状、环形与螺旋状等。用于创建笛卡儿拓扑的函数如下所示：

```
comm.Create_cart((number_of_rows,number_of_columns))
```

这里的 number_of_rows 与 number_of_columns 分别指定了将要创建的网格的行数与列数。

具体操作

在如下示例中，我们将会介绍如何实现一个大小为 $M \times N$ 的笛卡儿拓扑。此外，我们还定义了一组坐标来更好地理解所有进程是如何被处理的：

```
from mpi4py import MPI
import numpy as np
UP = 0
DOWN = 1
LEFT = 2
RIGHT = 3
neighbour_processes = [0,0,0,0]
if __name__ == "__main__":
    comm = MPI.COMM_WORLD
    rank = comm.rank
    size = comm.size

    grid_rows = int(np.floor(np.sqrt(comm.size)))
    grid_column = comm.size // grid_rows


    if grid_rows*grid_column > size:
        grid_column -= 1
    if grid_rows*grid_column > size:
```

```
        grid_rows -= 1

    if (rank == 0) :
        print("Building a %d x %d grid topology:"\
              % (grid_rows, grid_column) )


    cartesian_communicator = \
                           comm.Create_cart( \
                               (grid_rows, grid_column), \
                               periods=(True, True), reorder=True)
    my_mpi_row, my_mpi_col = \
                cartesian_communicator.Get_coords\
                ( cartesian_communicator.rank )

    neighbour_processes[UP], neighbour_processes[DOWN]\
                             = cartesian_communicator.Shift(0, 1)
    neighbour_processes[LEFT],  \
                               neighbour_processes[RIGHT]  = \
                               cartesian_communicator.Shift(1, 1)
    print ("Process = %s \
row = %s \
column = %s ----> neighbour_processes[UP] = %s \
neighbour_processes[DOWN] = %s \
neighbour_processes[LEFT] =%s neighbour_processes[RIGHT]=%s" \
           %(rank, my_mpi_row, \
             my_mpi_col,neighbour_processes[UP], \

neighbour_processes[DOWN], \
neighbour_processes[LEFT] , \
neighbour_processes[RIGHT]))
```

运行脚本后，输出结果如下所示：

```
C:\>mpiexec -n 4 python virtualTopology.py
Building a 2 x 2 grid topology:
Process = 0 row = 0 column = 0 ---->
neighbour_processes[UP] = -1
neighbour_processes[DOWN] = 2
neighbour_processes[LEFT] =-1
neighbour_processes[RIGHT]=1

Process = 1 row = 0 column = 1 ---->
neighbour_processes[UP] = -1
```

```
neighbour_processes[DOWN] = 3
neighbour_processes[LEFT] =0
neighbour_processes[RIGHT]=-1

Process = 2 row = 1 column = 0 ---->
neighbour_processes[UP] = 0
neighbour_processes[DOWN] = -1
neighbour_processes[LEFT] =-1
neighbour_processes[RIGHT]=3

Process = 3 row = 1 column = 1 ---->
neighbour_processes[UP] = 1
neighbour_processes[DOWN] = -1
neighbour_processes[LEFT] =2
neighbour_processes[RIGHT]=-1
```

对于每个进程来说，输出应该这样理解：如果 neighbour_processes = -1，那么它没有拓扑临近进程；否则，neighbour_processes 会显示出旁边进程的 rank 值。

实例精解

生成的拓扑是一个 2×2 的网状结构（参考之前的网状图），其大小等于输入的进程数，即 4：

```
grid_rows = int(np.floor(np.sqrt(comm.size)))
grid_column = comm.size // grid_rows
    if grid_rows*grid_column > size:
        grid_column -= 1
    if grid_rows*grid_column > size:
        grid_rows -= 1
```

接下来构建出笛卡儿拓扑：

```
cartesian_communicator = comm.Create_cart( \
     (grid_rows, grid_column), periods=(False, False), reorder=True)
...
```

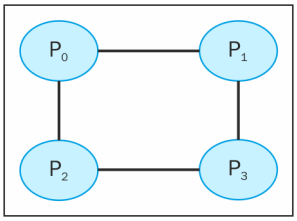
为了找出第 *i* 个进程的位置，我们以如下形式使用 Get_coords() 方法：

```
my_mpi_row, my_mpi_col = cartesian_communicator.Get_coords( cartesian_
communicator.rank )
For each process, in addition to their coordinates, we calculated
and got to know which processes are topologically closer. For
this purpose, we used the comm.Shift function comm.Shift (rank_
source,rank_dest)
```

根据该形式，我们有：

```
neighbour_processes[UP], neighbour_processes[DOWN] = \ cartesian_
communicator.Shift(0, 1)

neighbour_processes[LEFT], neighbour_processes[RIGHT] = \ cartesian_
communicator.Shift(1, 1)
```

所得到的拓扑展示在了下图中。



虚拟网状 2×2 拓扑

知识扩展

为了得到一个大小为 $M \times N$ 的螺旋拓扑，我们使用了如下代码：

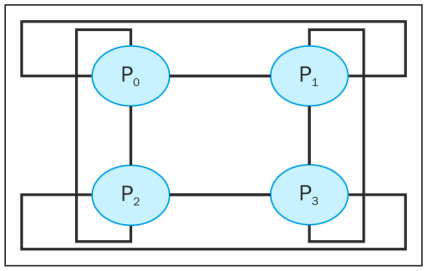
```
cartesian_communicator = comm.Create_cart( (grid_rows, grid_column),
periods=(True, True), reorder=True)
```

这会生成如下输出：

```
C:\>mpiexec -n 4 python VirtualTopology.py
Building a 2 x 2 grid topology:
Process = 0 row = 0 column = 0 ---->
neighbour_processes[UP] = 2
neighbour_processes[DOWN] = 2
neighbour_processes[LEFT] =1
neighbour_processes[RIGHT]=1
Process = 1 row = 0 column = 1 ---->
neighbour_processes[UP] = 3
neighbour_processes[DOWN] = 3
neighbour_processes[LEFT] =0
neighbour_processes[RIGHT]=0
Process = 2 row = 1 column = 0 ---->
neighbour_processes[UP] = 0
neighbour_processes[DOWN] = 0
neighbour_processes[LEFT] =3 neighbour_processes[RIGHT]=3
Process = 3 row = 1 column = 1 ---->
```

```
neighbour_processes[UP] = 1
neighbour_processes[DOWN] = 1
neighbour_processes[LEFT] =2
neighbour_processes[RIGHT]=2
```

它所表示的拓扑如下图所示。



虚拟的螺旋形 2×2 拓扑

4
异步编程

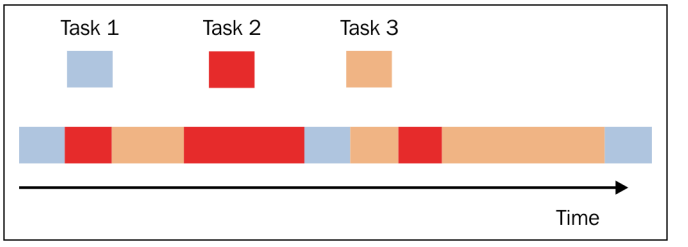
本章主要内容：

- 如何使用 Python 中的 `concurrent.futures` 模块
- 使用 Asyncio 实现事件循环管理
- 使用 Asyncio 处理协同程序
- 使用 Asyncio 管理任务
- 使用 Asyncio 和 Futures

介绍

除了线性和并行执行模式之外，还有一种被称为异步的模式，它与事件编程（event programming）一样，对我们来说至关重要。在单处理器系统和多处理器系统中，异步活动的执行模型均可使用一个主控制流来实现。

在并发执行的异步模式中，不同的任务在时间线上是相互交错的，而且一切都是在单一控制流（单线程）下进行的。一个任务的执行可以暂停，然后继续，不过这也改变了其他任务的执行时间。下图清晰地说明了这个概念。



异步编程模型

正如你所见，任务（用不同颜色表示）之间彼此交错，但都在同一个线程的控制之下；这意味着当执行某个任务时，其他任务没在执行。多线程编程模式与单线程异步并发模型的一个关键区别在于，前者由操作系统决定任务执行的时间线，即是否暂停某个线程的活动并开启另一个线程，而后者要求程序员假设某个线程可能被暂停，并随时被另一个线程取代。

程序员可以将任务编写为一系列可以间断式执行的小步骤；因此，如果某个任务需要使用另一个任务的输出，那么在编写该任务时就在程序中设定必须接受后者的输入。

使用 Python的 concurrent.futures 模块

随着 Python 3.2 版本的发布，Python 引入了 `concurrent.futures` 模块，支持管理并发编程任务，如进程池和线程池、非确定性执行流以及多进程和线程同步。

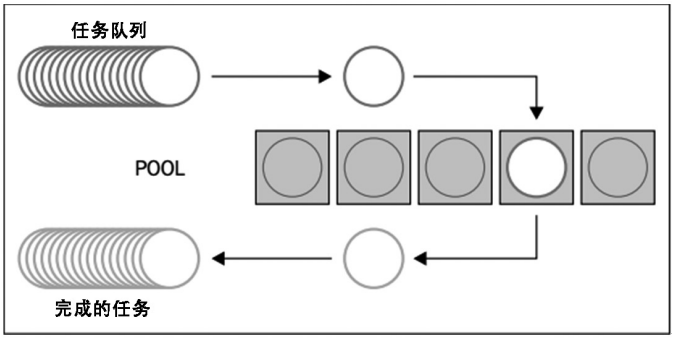
该模块包含以下几个类。

- `concurrent.futures.Executor`：这是一个抽象类，提供异步执行调用的方法。
- `submit(function, argument)`：安排某个函数（也叫可调用对象）使用给定参数执行。
- `map(function, argument)`：以异步模式使用给定参数来执行函数。
- `shutdown(Wait = True)`：向执行器（executor）传递释放资源的信号。
- `concurrent.futures.Future`：封装一个可调用函数的异步执行。可通过向执行器提交任务（带可选参数的函数）来实例化 Future 对象。

执行器是一种抽象类，通过其子类来访问：线程或进程的 `ExecutorPools`。实际上，实例化线程和进程是一个比较耗资源的任务，所以最好将这些资源聚集起来，把它们变成可重复使用的发射器（launcher）或执行器（执行器概念就是由此而来），用于并行或并发执行任务。

使用线程池和进程池

一个线程池或进程池（也被称为池化）指的是用来优化、简化程序内部线程 / 进程使用的软件管理器。通过池化，你可以向 pooler 提交将由其执行的任务。这个池子里有一个待执行任务的内部队列，以及一些执行这些任务的线程或进程，如下图所示。池化中的一个常见概念是复用：一个线程（或进程）在其生命周期中，被多次用于执行不同的任务。复用减少了创建进程或线程的开销，提升了利用池化技巧的程序的性能。虽然复用不是非用不可的，但它却是促使程序员在其应用中使用池化的主要原因之一。



池化管理

准备工作

current.Futures 模块提供了 Executor 类的两个子类，这两个子类分别可以异步式地管理一个线程池和一个进程池。它们是：

- concurrent.futures.ThreadPoolExecutor(max_workers)
- concurrent.futures.ProcessPoolExecutor(max_workers)

max_workers 参数表示用于异步执行调用的最大 worker 数量。

具体实现

下面的示例展示了线程池化和进程池化的作用。要执行的任务是：有一个由数字 1~10 组成的列表 number_list，针对列表中的每个元素，执行 1000 万次计数迭代（纯粹为了消磨时间），然后将得到的值与该元素相乘。

这样的话，可以清晰地比较以下两种情况：

- 线性执行
- 具备 5 个 worker 线程的线程池

请看下面的代码：

```
#
# Concurrent.Futures Pooling - 第 4 章：异步编程
#
import concurrent.futures
import time

number_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
def evaluate_item(x):
    # 计数，就是为了执行一些操作而已
    result_item = count(x)
    # 打印输入项及结果
    print("item " + str(x) + " result " + str(result_item))


def count(number):
    for i in range(0, 10000000):
        i = i+1
    return i*number


if __name__ == "__main__":

    ## 线性执行
    start_time = time.clock()
    for item in number_list:
        evaluate_item(item)
    print("Sequential execution in " +
          str(time.clock() - start_time), "seconds")

    ## 线程池执行
    start_time_1 = time.clock()
    with concurrent.futures.ThreadPoolExecutor(max_workers=5)\
          as executor:
      for item in number_list:
          executor.submit(evaluate_item, item)
    print("Thread pool execution in " +
        str(time.clock() - start_time_1), "seconds")

    ## 进程池执行
    start_time_2 = time.clock()
    with concurrent.futures.ProcessPoolExecutor(max_workers=5)\
            as executor:
           for item in number_list:
               executor.submit(evaluate_item, item)
        print("Process pool execution in " +
             str(time.clock() - start_time_2), "seconds")
```

运行上面的代码，我们将得到如下执行时间的结果：

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\ >python
Process_pool_with_concurrent_futures.py
item 1 result 10000000
item 2 result 20000000
```

```
item 3 result 30000000
item 4 result 40000000
item 5 result 50000000
item 6 result 60000000
item 7 result 70000000
item 8 result 80000000
item 9 result 90000000
item 10 result 100000000
Sequential execution in 17.241238674183425 seconds

Asynchronous Programming
item 4 result 40000000
item 2 result 20000000
item 1 result 10000000
item 5 result 50000000
item 3 result 30000000
item 7 result 70000000
item 6 result 60000000
item 8 result 80000000
item 10 result 100000000
item 9 result 90000000
Thread pool execution in 17.14648646290675 seconds

item 3 result 30000000
item 1 result 10000000
item 2 result 20000000
item 4 result 40000000
item 5 result 50000000
item 6 result 60000000
item 7 result 70000000
item 9 result 90000000
item 8 result 80000000
item 10 result 100000000
Process pool execution in 9.913172716938618 seconds
```

实例精解

我们首先创建一个数字列表，存储在 number_list 中。对于列表中的每个元素，执行计数程序，直到完成 1000 万次迭代。然后，我们再将得到的结果乘以 100 000 000：

```
def evaluate_item(x):
    # 计数，就是为了执行一些操作而已
    result_item = count(x)

def count(number):
    for i in range(0, 10000000):
```

```
        i = i+1
    return i*number
```

在主程序中，我们首先以线性模式执行任务：

```
if __name__ == "__main__":
    start_time = time.clock()
    for item in number_list:
        evaluate_item(item)
```

另外，在并发模式中，我们将使用 concurrent.futures 模块的池化功能，创建一个线程池：

```
with concurrent.futures.ThreadPoolExecutor(max_workers=5)\
      as executor:
  for item in number_list:
      executor.submit(evaluate_item, item)
```

ThreadPoolExecutor 使用其内部已经池化的线程执行给定的任务。它将管理在池子里工作的 5 个线程。每个线程从池子里接受并执行一个工作（job）。执行完工作后，线程将从线程池获取要处理的下一个工作。

当处理完所有工作后，打印执行时间：

```
print("Thread pool execution in " +
          str(time.clock() - start_time_1), "seconds")
```

要使用 ProcessPoolExecutor 类实现的进程池，可以这样写：

```
with concurrent.futures.ProcessPoolExecutor(max_workers=5)\
      as executor:
      for item in number_list:
        executor.submit(evaluate_item, item)
```

与 ThreadPoolExecutor 类似，ProcessPoolExecutor 是一个执行器子类，使用一个进程池异步执行调用。但是不同的是，ProcessPoolExecutor 使用的是 multiprocessing 模块，后者可以让我们避开全局解释器锁（global interpreter lock），大幅降低执行时间。

知识扩展

几乎所有服务器端应用都用到了池化，因为需要处理来自任意数量客户端的大量并发请求。而还有不少其他的应用要求每个任务立刻执行，或者对执行任务的线程具备更大的控制权。在这种情况下，池化不是最好的选择。

使用Asyncio实现事件循环管理

Python 模块 Asyncio 提供了管理事件、协程、任务和线程的功能，以及编写并发代码的同

步原语（synchronization primitives）。该模块主要由以下组件构成。

- **事件循（event loop）**：Asyncio 模块支持每个进程拥有一个事件循环。
- **协程（coroutines）**：这是子例程（subroutine）概念的泛化。另外，协程在执行时可以暂停，以等待外部处理程序完成（I/O 中的某个例行程序），外部处理程序结束后则从暂停之处返回。
- **Futures**：定义了 `Future` 对象，类似代表了尚未完成计算的 `concurrent.futures` 模块。
- **任务（tasks）**：这是 Asyncio 中的一个子类，用于封装并管理并行模式下的协程。

在这个实例中，重点是如何处理事件。事实上，事件在异步编程的语境下是非常重要的，因为其本质上就是异步的。

什么是事件循环

在计算系统中，能够产生事件的实体被称为事件源（event source），而负责协商管理事件的实体则被称为事件处理器（event handler）。有时可能还存在被称为事件循环的第三个实体。它实现了管理计算代码中所有事件的功能。更准确地说，在程序执行期间事件循环不断周期反复，追踪某个数据结构内部发生的事件，将其纳入队列，如果主线程空闲则调用事件处理器一个一个地处理这些事件。最后，我们来看一段事件循环管理器的伪代码。

```
while (1) {
events = getEvents();
for (e in events)
processEvent(e);
}
```

`while` 循环中的所有事件被事件处理器捕捉，然后逐一处理。事件的处理器是系统中唯一进行的活动。在处理器结束后，控制被传递给下一个执行的事件。

准备工作

Asyncio 提供了以下用于管理事件循环的方法。

- `loop = get_event_loop()`：使用该方法，可以获得当前上下文的事件循环。
- `loop.call_later(time_delay,callback,argument)`：该方法安排在给定时间 `time_delay` 秒后，调用某个回调对象。
- `loop.call_soon(callback,argument)`：该方法安排一个将马上被调用的回调对象。在 `call_soon()` 返回、控制回到事件循环之后，回调对象就被调用。
- `loop.time()`：以浮点值的形式返回根据事件循环的内部时钟确定的当前时间。
- `asyncio.set_event_loop()`：将当前上下文的事件循环设置为给定循环。
- `asyncio.new_event_loop()`：根据此函数的规则创建并返回一个新的事件循环对象。

- loop.run_forever()：一直执行，直到调用 stop() 为止。

具体实现

在本例中，我们将展示如何使用 Asyncio 库提供的循环事件语句，创建一个异步模式的应用。请看下面的代码：

```
import asyncio
import datetime
import time


def function_1(end_time, loop):
    print("function_1 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_2, end_time, loop)
    else:
        loop.stop()


def function_2(end_time, loop):
    print("function_2 called ")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_3, end_time, loop)
    else:
        loop.stop()


def function_3(end_time, loop):
    print("function_3 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_1, end_time, loop)
    else:
        loop.stop()


def function_4(end_time, loop):
    print("function_5 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_4, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

end_loop = loop.time() + 9.0
```

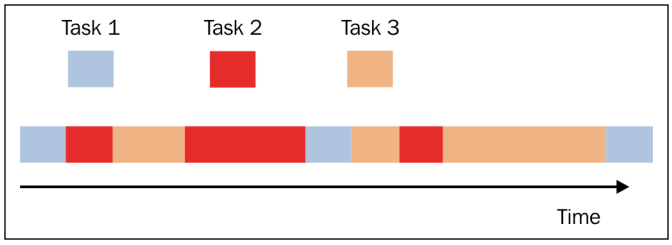
```
loop.call_soon(function_1, end_loop, loop)
# loop.call_soon(function_4, end_loop, loop)

loop.run_forever()
loop.close()
```

以上代码的输出如下所示：

```
C:\Python Parallel Programming INDEX\Chapter 4- Asynchronous
Programming >python asyncio_loop.py
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
```

实例精解

在此例中，我们定义了三个异步任务，各自依次调用下一个任务，如下图所示。



此例中的任务执行

为了实现各自依次调用，我们需要捕获整个事件循环：

```
loop = asyncio.get_event_loop()
```

然后，我们通过 call_soon 安排对 function_1() 的第一次调用：

```
end_loop = loop.time() + 9.0
loop.call_soon(function_1, end_loop, loop)
```

请注意 function_1 的定义：

```
def function_1(end_time, loop):
```

```
    print ("function_1 called")
    if (loop.time() + 1.0) < end_time:
       loop.call_later(1, function_2, end_time, loop)
    else:
        loop.stop()
```

上面的代码通过以下参数定义了程序的异步行为。

- end_time：定义了 function_1 内部的时间上限，并通过 call_later 方法调用 function_2。
- loop：这是此前通过 get_event_loop() 方法捕获的事件循环。

function_1 的任务非常简单，就是打印自己的名称，但也可以是更复杂的计算：

```
print ("function_1 called")
```

执行完任务后，将 loop.time() 与此次执行的耗时相比；总的周期数为 12，如果没有达到这个数目，则通过 call_later 方法继续执行，并延时 1 秒：

```
if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_2, end_time, loop)
    else:
        loop.stop()
```

function_2() 和 function_3() 执行同样的操作。

如果用完全部执行时间，循环事件必须终止：

```
loop.run_forever()
loop.close()
```

使用Asyncio处理协程

从上面介绍的多个示例中，我们看到：当程序变得冗长复杂时，将其划分成子例程的方式会使处理变得更加便利，每个子例程完成一个特定的任务，并针对任务实现合适的算法。子例程无法独立运行，只能在主程序的要求下才能运行，主程序负责协调子例程的使用。协程就是子例程的泛化。与子例程类似，协程执行一个计算步骤，但不同的是，不存在可用于协调结果的主程序。这是因为协程之间可以相互连接在一起，形成一个管道，不需要任何监督式函数来按顺序调用协程。在协程中，可以暂停执行点，同时保存干预时的本地状态，便于后续继续执行。有了协程池之后，协程计算就能够相互交错：运行第一个协程，直到其返回控制权，然后运行第二个协程，依此类推。

协程相互交错的控制组件就是事件循环，在上一个示例中我们已经介绍过了。事件循环追踪全部的协程，并安排其执行的时间。

协程的其他重要特点包括以下方面：

- 协程支持多个进入点，可以多次生成（yield）。
- 协程能够将执行转移至任何其他协程。

“生成”（yield）这个术语用于描述那些暂停并将控制流传递给另一个协程的协程。由于协程可以同时传递控制流和值，“生成一个值”（yielding a value）这个短语也用于描述生成并将值传递给获得控制流的协程。

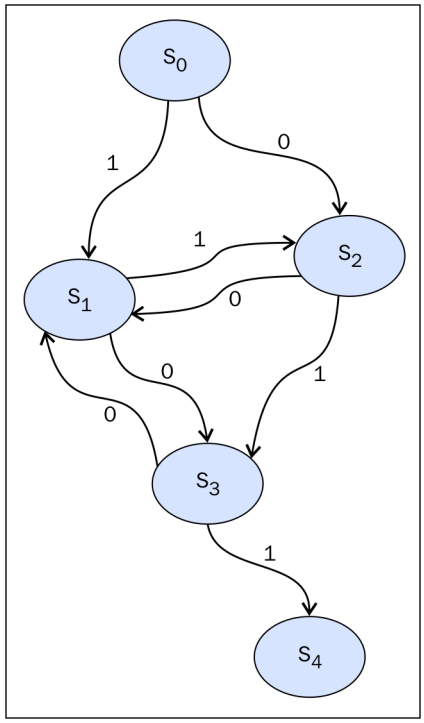
准备工作

利用 Asyncio 模块定义协程，我们只需要使用注释即可：

```
import asyncio
@asyncio.coroutine
def coroutine_function( function_arguments ) :
   # DO_SOMETHING
```

具体实现

在此例中，我们将会看到如何使用 Asyncio 的协程机制模拟一个具备 5 个状态的有限状态机。*有限状态机或自动机*（finite state machine or automaton，简称 FSA）这个数学模型不仅广泛用于工程学科，也普遍应用于基础科学，如数学和计算机科学。我们希望模拟的自动机（automata）的行为如右图所示。



有限状态机

在上面的图中，我们标明了系统的 5 个状态：S_0、S_1、S_2、S_3、S_4。这里，0 至 4 是自动机能够从一个状态传递至下一个状态的值（该操作被称为切换，transition）。例如，状态 S_0 只有在值为 1 时才能切换至 S_1，S_0 只有在值为 0 时才能切换至 S_2。下面的 Python 代码模拟了自动机从所谓的初始状态 S_0 切换至结束状态 S_4：

```
# Asyncio 有限状态机
import asyncio
import time
from random import randint


@asyncio.coroutine
def StartState():
    print("Start State called \n")
    input_value = randint(0, 1)
    time.sleep(1)
    if (input_value == 0):
        result = yield from State2(input_value)
    else:
        result = yield from State1(input_value)
    print("Resume of the Transition : \nStart State calling "
          + result)


@asyncio.coroutine
def State1(transition_value):
    outputValue = str(("State 1 with transition value = %s \n"
                       % (transition_value)))
    input_value = randint(0, 1)
    time.sleep(1)
    print("...Evaluating...")
    if (input_value == 0):
        result = yield from State3(input_value)
    else:
        result = yield from State2(input_value)
    result = "State 1 calling " + result
    return (outputValue + str(result))


@asyncio.coroutine
def State2(transition_value):
    outputValue = str(("State 2 with transition value = %s \n"
                       % (transition_value)))
    input_value = randint(0, 1)
    time.sleep(1)
```

```
    print("...Evaluating...")
    if (input_value == 0):
        result = yield from State1(input_value)
    else:
        result = yield from State3(input_value)
    result = "State 2 calling " + result
    return (outputValue + str(result))


@asyncio.coroutine
def State3(transition_value):
    outputValue = str(("State 3 with transition value = %s \n"
                       % (transition_value)))
    input_value = randint(0, 1)
    time.sleep(1)
    print("...Evaluating...")
    if (input_value == 0):
        result = yield from State1(input_value)
    else:
        result = yield from EndState(input_value)
    result = "State 3 calling " + result
    return (outputValue + str(result))


@asyncio.coroutine
def EndState(transition_value):
    outputValue = str(("End State with transition value = %s \n"
                       % (transition_value)))
    print("...Stop Computation...")
    return (outputValue)

if __name__ == "__main__":
    print("Finite State Machine simulation with Asyncio Coroutine")
    loop = asyncio.get_event_loop()
    loop.run_until_complete(StartState())
```

运行上述代码之后，我们会得到类似下面的输出：

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\codes - Chapter
4>python asyncio_state_machine.py
Finite State Machine simulation with Asyncio Coroutine
Start State called
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
```

```
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Evaluating...
...Stop Computation...
Resume of the Transition :
Start State calling State 1 with transition value = 1
State 1 calling State 3 with transition value = 0
State 3 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 3 with transition value = 1
State 3 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 1 with transition value = 0
State 1 calling State 3 with transition value = 0
State 3 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 3 with transition value = 1
State 3 calling End State with transition value = 1
```

实例精解

自动机的每个状态通过以下方式定义：

```
@asyncio.coroutine
```

例如，状态 S_0 是这样被定义的：

```
@asyncio.coroutine
def StartState():
    print ("Start State called \n")
    input_value = randint(0,1)
    time.sleep(1)
    if (input_value == 0):
        result = yield from State2(input_value)
    else :
        result = yield from State1(input_value)
```

是否切换至下一个状态由 input_value 决定，而它是由 Python 的 random 模块中的 randint(0,1) 函数定义的。该函数随机返回值 0 或 1。通过这种方法，可以随机决定有限状态机将会被传递哪个状态：

```
input_value = randint(0,1)
```

决定了有限状态机在什么值下切换状态后，协程使用 yield from 命令调用下一个协程：

```
if (input_value == 0):
        result = yield from State2(input_value)
   else :
        result = yield from State1(input_value)
```

变量 result 是每个协程返回的值。它是一个字符串，在计算结束时，我们可以重构自动机从初始状态切换到结束状态的全过程。

主程序在事件循环内部发起计算过程，如下所示：

```
if __name__ == "__main__":
   print("Finite State Machine simulation with Asyncio Coroutine")
   loop = asyncio.get_event_loop()
   loop.run_until_complete(StartState())
```

使用Asyncio管理任务

Asyncio 的宗旨是处理事件循环中的异步进程和并发执行任务。它还提供了一个叫作 asyncio.Task() 的类，用于将协程封装在任务中。该类的用途在于，支持独立运行的任务与同一个事件循环中的其他任务并发运行。协程被封装进任务中后，它将该任务与事件循环相连接，并在循环开始时自动运行，因此算是提供了一种自动驱动协程的机制。

准备工作

Asyncio 模块提供了一个处理任务计算的方法，asyncio.Task(coroutine)。该方法用于调度协程的执行。任务负责执行事件循环中的协程对象。如果被封装的协程从 future 生成，任务将暂停执行被封装的协程，并等待 future 执行完毕。

在 future 执行完毕后，被封装的协程以 future 返回的结果或异常重新开始执行。另外，我们必须注意，一个事件循环一次只执行一个任务。如果其他事件循环通过不同的线程运行，则可以并行执行其他任务。在任务等待 future 执行完毕时，事件循环将执行一个新任务。

具体实现

在下面的示例代码中，我们将介绍如何通过 Asyncio.Task() 语句并发执行三个数学函数：

```
"""
通过 Asyncio.Task 并发执行三个数学函数
"""
import asyncio
```

```
@asyncio.coroutine
def factorial(number):
    f = 1
    for i in range(2, number+1):
        print("Asyncio.Task: Compute factorial(%s)" % (i))
        yield from asyncio.sleep(1)
        f *= i
    print("Asyncio.Task - factorial(%s) = %s" % (number, f))


@asyncio.coroutine
def fibonacci(number):
    a, b = 0, 1
    for i in range(number):
        print("Asyncio.Task: Compute fibonacci (%s)" % (i))
        yield from asyncio.sleep(1)
        a, b = b, a + b
    print("Asyncio.Task - fibonacci(%s) = %s" % (number, a))

@asyncio.coroutine
def binomialCoeff(n, k):
    result = 1
    for i in range(1, k+1):
           result = result * (n-i+1) / i
           print("Asyncio.Task: Compute binomialCoeff (%s)" % (i))
           yield from asyncio.sleep(1)
    print("Asyncio.Task - binomialCoeff(%s , %s) = \
                    %s" % (n,k,result))

if __name__ == "__main__":
   tasks = [asyncio.Task(factorial(10)),
            asyncio.Task(fibonacci(10)),
            asyncio.Task(binomialCoeff(20,10))]
   loop = asyncio.get_event_loop()
   loop.run_until_complete(asyncio.wait(tasks))
   loop.close()
```

上述代码的输出结果是：

```
C:\ Python CookBook \Chapter 4- Asynchronous Programming\codes -
Chapter 4> python asyncio_Task.py
Asyncio.Task: Compute factorial(2)
Asyncio.Task: Compute fibonacci (0)
Asyncio.Task: Compute binomialCoeff (1)
```

```
Asyncio.Task: Compute factorial(3)
Asyncio.Task: Compute fibonacci (1)
Asyncio.Task: Compute binomialCoeff (2)
Asyncio.Task: Compute factorial(4)
Asyncio.Task: Compute fibonacci (2)
Asyncio.Task: Compute binomialCoeff (3)
Asyncio.Task: Compute factorial(5)
Asyncio.Task: Compute fibonacci (3)
Asyncio.Task: Compute binomialCoeff (4)
Asyncio.Task: Compute factorial(6)
Asyncio.Task: Compute fibonacci (4)
Asyncio.Task: Compute binomialCoeff (5)
Asyncio.Task: Compute factorial(7)
Asyncio.Task: Compute fibonacci (5)
Asyncio.Task: Compute binomialCoeff (6)
Asyncio.Task: Compute factorial(8)
Asyncio.Task: Compute fibonacci (6)
Asyncio.Task: Compute binomialCoeff (7)
Asyncio.Task: Compute factorial(9)
Asyncio.Task: Compute fibonacci (7)
Asyncio.Task: Compute binomialCoeff (8)
Asyncio.Task: Compute factorial(10)
Asyncio.Task: Compute fibonacci (8)
Asyncio.Task: Compute binomialCoeff (9)
Asyncio.Task - factorial(10) = 3628800
Asyncio.Task: Compute fibonacci (9)
Asyncio.Task: Compute binomialCoeff (10)
Asyncio.Task - fibonacci(10) = 55
Asyncio.Task - binomialCoeff(20 , 10) = 184756.0
```

实例精解

在此例中，我们定义了三个协程，factorial 、fibonacci 和 binomialCoeff，如前面解释的那样，每个都带有 @asyncio.coroutine 装饰器。

```
@asyncio.coroutine
def factorial(number):
do Something

@asyncio.coroutine
def fibonacci(number):
do Something

@asyncio.coroutine
def binomialCoeff(n, k):
do Something
```

如果要并行执行这三个任务，首先要将它们放入一个任务列表中，如下所示：

```
if __name__ == "__main__":
    tasks = [asyncio.Task(factorial(10)),
             asyncio.Task(fibonacci(10)),
             asyncio.Task(binomialCoeff(20,10))]
```

然后，获取 event_loop：

```
loop = asyncio.get_event_loop()
```

接下来，运行任务：

```
loop.run_until_complete(asyncio.wait(tasks))
```

这里，asyncio.wait(tasks) 将等待给定协程执行完毕。

在最后一个语句中，我们关闭事件循环：

```
loop.close()
```

使用Asyncio和Futures

Asyncio 模块的另一个关键组件是 Future 类。它与 concurrent.futures.Futures 非常相似，但是已经按照 Asyncio 的事件循环的主机制进行了调整。asyncio.Future 类代表一个还不可用的结果（但也可能是一个异常）。因此，它是对尚需完成的任务的抽象表示。

事实上，那些必须处理结果的回调对象也被算作该类的实例。

准备工作

我们必须做出如下定义才能管理 Asyncio 中的 Future 对象：

```
import asyncio
future = asyncio.Future()
```

该类具备如下基本方法。

- cancel()：取消 future，并安排回调对象。
- result()：返回 future 所代表的结果。
- exception()：返回 future 上设置的异常。
- add_done_callback(fn)：添加一个在 future 执行时运行的回调对象。
- remove_done_callback(fn)：从“结束后调用（call when done）”列表中移除一个回调对象的所有实例。

- set_result(result)：将 future 标记为已完成，并设置其结果。
- set_exception(exception)：将 future 标记为已完成，并设置一个异常。

具体实现

下面的例子演示了如何利用 Futures 类管理两个执行任务的协程，first_coroutines 和 second_coroutines。例如，这两个协程分别执行求前 *n* 个整数之和和计算 *n* 的阶乘的任务。代码如下：

```
"""
Asyncio.Futures -  第 4 章：异步编程
"""
import asyncio
import sys

# 求 n 个整数的和
@asyncio.coroutine
def first_coroutine(future, N):
    count = 0
    for i in range(1, N+1):
        count = count + i
    yield from asyncio.sleep(4)
    future.set_result("first coroutine (sum of N integers) result = "
                      + str(count))


# 求 n 的阶乘
@asyncio.coroutine
def second_coroutine(future, N):
    count = 1
    for i in range(2, N+1):
        count *= i
    yield from asyncio.sleep(3)
    future.set_result("second coroutine (factorial) result = "
                      + str(count))


def got_result(future):
    print(future.result())

if __name__ == "__main__":
    N1 = int(sys.argv[1])
    N2 = int(sys.argv[2])

    loop = asyncio.get_event_loop()
```

```
    future1 = asyncio.Future()
    future2 = asyncio.Future()

    tasks = [
        first_coroutine(future1, N1),
        second_coroutine(future2, N2)]

    future1.add_done_callback(got_result)
    future2.add_done_callback(got_result)

    loop.run_until_complete(asyncio.wait(tasks))
    loop.close()
```

运行上述代码之后，我们获得了类似下面的输出：

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\codes - Chapter
4>python Asyncio_future.py 1 1
first coroutine (sum of N integers) result = 1
second coroutine (factorial) result = 1

C:\Python CookBook\Chapter 4- Asynchronous Programming\codes - Chapter
4>python Asyncio_future.py 2 2
first coroutine (sum of N integers) result = 3
second coroutine (factorial) result = 2

C:\ Python CookBook\Chapter 4- Asynchronous Programming\codes -
Chapter 4>python Asyncio_future.py 3 3
first coroutine (sum of N integers) result = 6
second coroutine (factorial) result = 6

C:\ Python CookBook\Chapter 4- Asynchronous Programming\codes -
Chapter 4>python Asyncio_future.py 5 5
first coroutine (sum of N integers) result = 15
second coroutine (factorial) result = 120
```

实例精解

在主程序中，我们定义两个 future 对象，与协程关联在一起。

```
if __name__ == "__main__":

future1 = asyncio.Future()
future2 = asyncio.Future()
```

在定义任务时，我们将 future 对象作为协程的实参传入：

```
tasks = [first_coroutine(future1, N1),
         second_coroutine(future2, N2)]
```

最后，我们添加一个 future 执行时将运行的回调对象：

```
future1.add_done_callback(got_result)
future2.add_done_callback(got_result)
```

这里，got_result 是一个打印 future 最后结果的函数。

```
def got_result(future):
    print(future.result())
```

在传入 future 对象作为实参的协程中，完成计算之后，我们为第一个协程设置 3 秒的睡眠时间，为第二个协程设置 4 秒的睡眠时间。

```
yield from asyncio.sleep(sleep_time)
```

然后，标记 future 为已完成，在 future.set_result() 的帮助下设置其结果。

知识扩展

将协程之间的睡眠时间对调后，我们可以将输出结果的顺序颠倒过来（先对第二个协程的输出进行这样的操作）：

```
C:\Python CookBook\Chapter 4- Asynchronous Programming\codes - Chapter
4>python Asyncio_future.py 1 10
second coroutine (factorial) result = 3628800
first coroutine (sum of N integers) result = 1
```

5
分布式Python

本章主要内容：

- 使用 Celery 分发任务
- 如何使用 Celery 创建任务
- 使用 SCOOP 进行科学计算
- 使用 SCOOP 处理映射函数
- 使用 Pyro4 远程调用方法
- 使用 Pyro4 链接对象
- 使用 Pyro4 开发一个客户端 - 服务器应用
- 使用 PyCSP 实现顺序进程的通信
- 在 Disco 中使用 MapReduce
- 使用 RPyC 调用远程过程

介绍

分布式计算的基本理念是将工作划分为一个一个的小任务，通常每个任务都有名称。分布式网络中的电脑可以完美无缺地完成这些合理大小的任务并返回结果。在分布式计算中，网络中的机器必须要保持可用（延迟误差、意外宕机或者电脑联网，等等）。因此，你需要一个持续监控架构。

使用这种技术所带来的根本问题，主要在于对（传输与接收没有出错的）流量（数据、工作、命令等）的合理管理。另外，还有一个源自分布式计算基本特征的问题：分布式网络中共存的机器支持不同的操作系统，而这些系统通常相互不兼容。事实上，由于需要在分布式环境中使用多种多样的资源，逐渐出现了不同的计算模型。它们的目标基本都是为如何描述分布式应用的进程之间的协作提供框架。我们可以这样说，不同模型的区别基本在于其对分布式所提供机

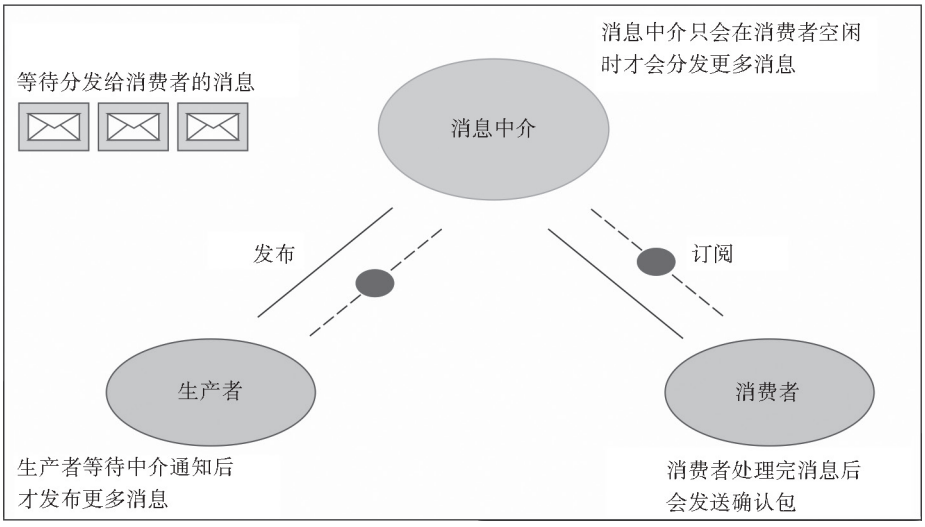
会的利用能力。使用最广泛的模型是客户端 - 服务器（client-server）模型。它可以让位于不同电脑中的进程通过交换信息实现实时协作，因此性能比之前的模型有很大提升，后者要求转移所有的文件，而且要离线对数据进行计算。客户端 - 服务器模型通常通过远程进程调用（这将扩大本机调用的范围），或分布式对象范式（面向对象中间件）实现。本章将介绍 Python 中实现这些计算架构的部分方案，然后探讨使用面向对象模式和远程调用模式实现分布式架构的库，如 Celery、SCOOP、Pyro4 和 RPyC，以及使用其他不同方法实现的库，如 PyCSP 和 Disco，后两者是 Python 中对应于 MapReduce 的算法。

使用 Celery 分发任务

Celery 是一个用于管理分布式任务的 Python 框架，采用的是面向对象中间件的方法实现。其主要特性包括处理大量小型任务，并将其分发给大量计算节点。最后，每个任务的结果重新组合，构成最终的答案。

要想使用 Celery，我们需要以下组件。

- Celery 模块（当然是必需的）。
- 消息中介（message broker）。这是一个不依赖于 Celery 的软件组件，是一个中间件，用于向分布式任务工作进程发送和接收信息，如下图所示。消息中介也被称为消息中间件。它负责通信网络中的消息交换。这类中间件的编址方案（addressing scheme）不再是点对点式的，而是面向消息式的。其中最知名的就是发布 / 订阅范式。



消息中介架构

Celery 支持多种类型的消息中介，其中最为完整的是 RabbitMQ 和 Redis。

具体实现

我们使用 pip 安装 Celery。在命令提示符中，输入以下命令：

```
pip install celery
```

然后，我们必须安装消息中介。这里有多个选择，但在本书的示例中，我们使用的是 RabbitMQ，它是一个面向消息的中间件（也被称为中介消息传递，broker messaging），它实现了高级消息队列协议（Advanced Message Queuing Protocol，简称 AMQP）。RabbitMQ 服务器采用 Erlang 语言编写，其基础是用于管理集群和故障切换的开放电信平台（Open Telecom Platform，简称 OTP）框架。要想安装 RabbitMQ，需先下载并运行 Erlang（http://www.erlang.org/download.html），然后再下载并运行 RabbitMQ 安装器即可（http://www.rabbitmq.com/download.html）。RabbitMQ 的安装和配置需要花一点时间，之后以默认配置将其作为系统服务运行。

最后，我们安装 Flower（http://flower.readthedocs.org），它是一个用于监控任务（运行进度、任务详情和图表及数据）的 Web 工具。

在命令提示符中输入以下命令，安装 Flower：

```
pip install -U flower
```

然后，我们即可验证 Celery 是否安装成功。在命令提示符中，输入以下命令：

```
C:\celery --version
```

之后应该会出现类似下面的文字：

```
3.1.18 (Cipater)
```

Celery 的使用非常简单，如下所示：

```
Usage: celery <command> [options]
```

这里，可选项如下所示：

```
Options:
     -A APP, --app=APP     app instance to use (e.g. module.attr_name)
     -b BROKER, --broker=BROKER
                           url to broker.  default is 'amqp://guest@
                           localhost//'
     --loader=LOADER       name of custom loader class to use.
     --config=CONFIG       Name of the configuration module
     --workdir=WORKING_DIRECTORY
                           Optional directory to change to after detaching.
     -C, --no-color
     -q, --quiet
```

```
    --version             show program's version number and exit
    -h, --help            show this help message and exit
```

参考

- 如想了解 Celery 安装过程的更多细节，可访问 www.celeryproject.com。

如何使用 Celery 创建任务

在该示例中，我们将介绍如何使用 Celery 模块创建并调用任务。Celery 提供了以下调用任务的方法。

- apply_async(args[, kwargs[, ...]])：该任务发送一个任务消息。
- delay(*args, **kwargs)：这是发送任务消息的便捷方法，但不支持添加执行选项。

delay 方法更好用，因为它可以像普通函数一样被调用：

```
task.delay(arg1, arg2, kwarg1='x', kwarg2='y')
```

但使用 apply_async 的话，你应该这样写：

```
task.apply_async (args=[arg1, arg2], kwargs={'kwarg1': 'x','kwarg2': 'y'})
```

具体实现

如想执行一个简单的任务，我们需实现以下两个小脚本：

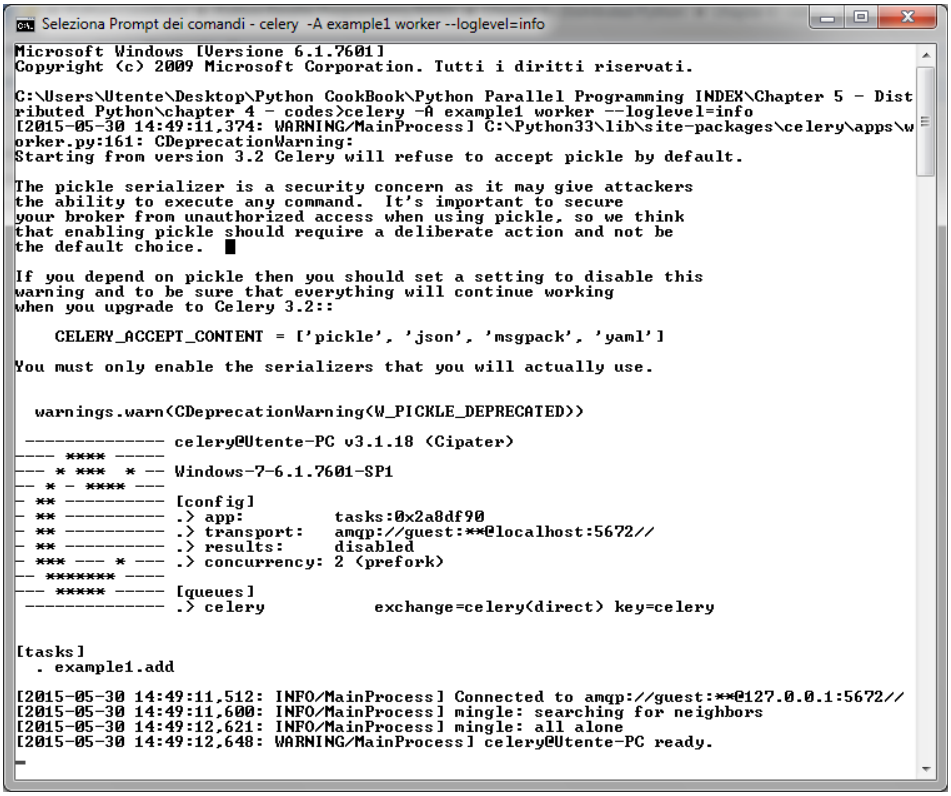
```
###
## addTask.py ：执行一个简单任务
###

from celery import Celery

app = Celery('addTask',broker='amqp://guest@localhost//')

@app.task
def add(x, y):
   return x + y

# 第二个脚本为：

###
#addTask.py : 运行 addTask 示例
###
import addTask
```

```
if __name__ == '__main__':
    result = addTask.add.delay(5,5)
```

再次提醒一下，RabbitMQ 服务在安装完成后会在服务器上自动运行。因此，如要执行 Celery 工作者服务器，只需在命令提示符中输入以下命令：

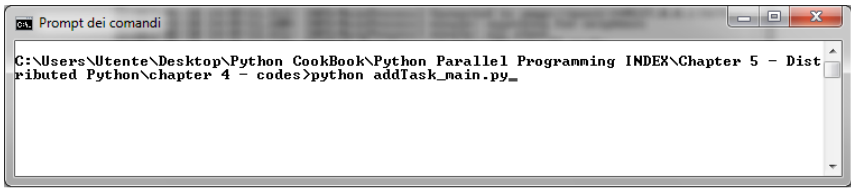
```
celery -A addTask worker --loglevel=info
```

第一个命令提示符中的输出如下图所示。

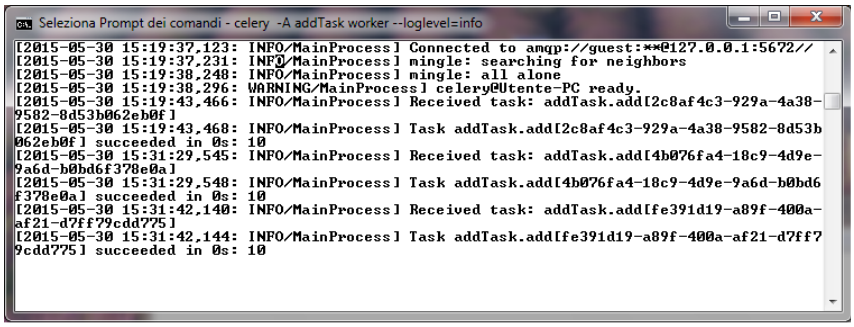


请注意输出中的警告信息，建议出于安全考虑，禁用 pickle 作为序列化器（serializer）。默认的序列化格式之所以是 pickle，只是因为它方便（支持将复杂的 Python 对象作为任务参数传入）。无论你是否使用 pickle，都应该设置 CELERY_ACCEPT_CONTENT 配置变量，关闭这一警告信息；可以查看 http://celery.readthedocs.org/en/latest/configuration.html 作为参考。

现在，我们从另一个命令提示符中启动 addTask_main 脚本，如下图所示。



最后，第一个命令提示符中的输出结果应该类似下图所示。



结果是 10（从最后一行输出可知），和预料的一样。

实例精解

我们先来看第一个脚本，addTask.py。在前两行代码中，创建了一个 Celery 应用实例，其使用了 RabbitMQ 服务和中介：

```
from celery import Celery
app = Celery('addTask', broker='amqp://guest@localhost//')
```

Celery 函数的第一个实参是当前模块的名称（addTask.py），第二个实参是中介参数，指出了用于连接中介（RabbitMQ）的 URL 。然后，我们引入任务。每个任务必须添加 @app.task 注释。

装饰器帮助 Celery 辨别任务队列中的哪些函数可以调度。在装饰器之下，我们创建工作者进程可以执行的任务。第一个任务是求两个数字之和的简单函数：

```
@app.task
def add(x, y):
    return x + y
```

在第二个脚本 AddTask_main.py 中，使用 delay() 方法调用该任务：

```
if __name__ == '__main__':
   result = addTask.add.delay(5,5)
```

记住，这个方法是 apply_async() 方法的快捷方式，可让我们更好地控制任务执行。

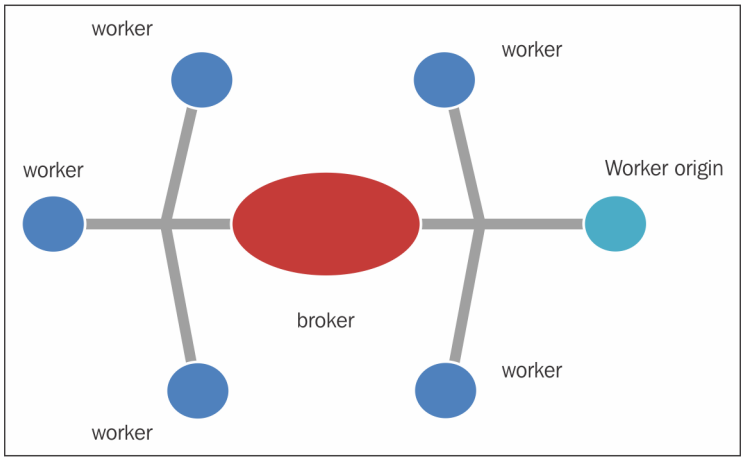
知识扩展

如果以默认配置运行 RabbitMQ，Celery 则只能连接 amqp://scheme 这样的地址。

使用 SCOOP 进行科学计算

SCOOP，全称 Scalable Concurrent Operations in Python，意为“Python 中的可伸缩式并发运算”，是一个将并发任务（被称为 Futures）分发在异构计算节点（heterogeneous computational nodes）上运行的 Python 模块。其架构基于 ØMQ 包，后者提供了在分布式系统之间管理 Futures 的方法。SCOOP 应用在需要使用全部可用计算资源执行诸多分布式任务的科学计算。

SCOOP 采用了下图所示的这种中介模式来分发 Futures。



SCOOP 架构

通信系统的核心元素是负责与所有独立工作者进程交互、调度消息的中介。Futures 并非创建于中心节点（中介），而是通过一套统一的序列化程序在各个工作者进程中创建。这使得其拓扑架构更加可靠，性能更高。实际上，中介的主要工作负载由网络和工作者进程间 I/O 构成，所需的 CPU 处理时间相对较少。

准备工作

SCOOP 模块可在 https://github.com/soravux/scoop/ 处获取，所需依赖如下：

- Python >= 2.6 或 >= 3.2

- Distribute >= 0.6.2 或 setuptools >= 0.7
- Greenlet >= 0.3.4
- pyzmq >= 13.1.0 及 libzmq >= 3.2.0
- SSH（用于远程执行）

SCOOP 可安装于 Linux、Mac 和 Windows 机器中。和 Disco 一样，远程使用 SCOOP 需要 SSH 软件，而且各个节点之间必须开启为无须密码验证。有关 SCOOP 安装过程的完整参考，请参阅 http://scoop.readthedocs.org/en/0.7/install.html 中的信息指南。

在 Windows 机器中，输入以下命令即可安装 SCOOP：

```
pip install SCOOP
```

另外，也可以从 SCOOP 发布版的目录下输入以下命令：

```
Python setup.py install
```

具体实现

SCOOP 库具备许多功能，主要用于科学计算。尽管求解科学计算问题的方法十分耗费计算资源，但幸好出现了蒙特卡罗算法。关于此方法的完整论述可能会占据本书很大篇幅，但是在此例中，我们打算介绍如何使用 SCOOP 的特性并行化蒙特卡罗方法求解以下问题：计算数字 π。因此，让我们来看以下代码：

```
import math
from random import random
from scoop import futures
from time import time


def evaluate_number_of_points_in_unit_circle(attempts):
    points_fallen_in_unit_disk = 0
    for i in range(0, attempts):
        x = random()
        y = random()
        radius = math.sqrt(x*x + y*y)
        # 如果点落在单位圆内，则测试通过
        if radius < 1:
            # 测试通过后，增加圆内点的数目
            points_fallen_in_unit_disk = \
                points_fallen_in_unit_disk + 1
    return points_fallen_in_unit_disk


def pi_calculus_with_Montecarlo_Method(workers, attempts):
```

```
    print("number of workers %i - number of attempts %i"
          % (workers, attempts))
    bt = time()
    # 这时我们调用 scoop.futures.map 函数
    # 然后求单位圆内点的数目
    # 函数以异步方式执行
    # 并且可以并发多次调用该函数
    evaluate_task = \
        futures.map(evaluate_points_in_circle,
                    [attempts] * workers)
    taskresult = sum(evaluate_task)
    print("%i points fallen in a unit disk after "
          % (Taskresult/attempts))
    piValue = (4. * Taskresult / float(workers * attempts))

    computationalTime = time() - bt
    print("value of pi = " + str(piValue))
    print("error percentage = " +
          str((((abs(piValue - math.pi)) * 100) / math.pi)))
    print("total time: " + str(computationalTime))

if __name__ == "__main__":
    for i in range(1, 4):
        # 我们将工作者数量固定为 2,
        # 当然数量也可以设置得更高
        pi_calculus_with_Montecarlo_Method(i*1000, i*1000)
        print(" ")
```

如要运行一个 SCOOP 程序，必须打开命令提示符并输入类似下面的指令：

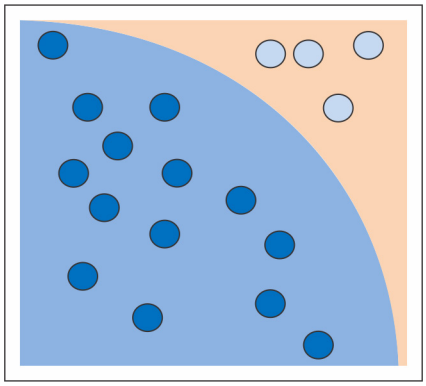
```
python -m scoop name_file.py
```

运行脚本之后，应该会出现如下输出：

```
C:\Python CookBook\Chapter 5 - Distributed Python\chapter 5 -
codes>python -m scoop pi_calculus_with_montecarlo_method.py
[2015-06-01 15:16:32,685] launcher  INFO   SCOOP 0.7.2 dev on win32
using Python 3.3.0 (v3.3.0:bd8afb90e
bf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)], API: 1013
[2015-06-01 15:16:32,685] launcher   INFO     Deploying 2 worker(s) over 1
host(s).
[2015-06-01 15:16:32,685] launcher  INFO    Worker d--istribution:
[2015-06-01 15:16:32,686] launcher  INFO        127.0.0.1:        1 + origin
Launching 2 worker(s) using an unknown shell.
number of workers 1000 - number of attempts 1000
785 points fallen in a unit disk after
value of pi = 3.140636
```

```
error percentage = 0.030451229528429 62
total time: 10.258585929870605


number of workers 2000 - number of attempts 2000
1570 points fallen in a unit disk after
value of pi = 3.141976
error percentage = 0.012202295220195048
total time: 20.451170206069946


number of workers 3000 - number of attempts 3000
2356 points fallen in a unit disk after
value of pi = 3.1413777777777776
error percentage = 0.006839709526630775
total time: 32.3558509349823


[2015-06-01 15:17:36,894] launcher   (127.0.0.1:59239) INFO    Root process
is done.
[2015-06-01 15:17:36,896] launcher   (127.0.0.1:59239) INFO    Finished
cleaning spawned subprocesses.
```

随着尝试次数和工作者数量不断增加，π 的正确值也越来越精确，如下图所示。



蒙特卡罗算法求 π 的值：计算圆内的点数

实例精解

上一节给出的代码只是诸多求解 π 值的蒙特卡罗算法实现中的一个。随机性地执行 `evaluate_ points_in_circle()` 函数，然后给定一个坐标点 (x,y)，并判断该点是否位于单位区域所在的圆内。

一旦 points_fallen_in_unit_disk 条件被验证成立，那么对应的变量加 1。当函数的内部循环结束时，该变量也就代表了落入圆内的点数。这个数字足够用于计算 π 的值。事实上，点落入圆内的概率是 π/4，也就是单位圆的面积（等于 π）与其外接正方形面积（等于 4）的比。

因此，通过计算落入半圆内的点数（taskresult）与总尝试次数（工作者数量 × 尝试次数）直接的比例，即可获得 π/4 的近似值，当然也就求得了数字 π 的值：

```
piValue = (  4. * taskresult / float (workers attempts *))
```

SCOOP 函数如下所示：

```
futures.map (evaluate_points_in_circle, [attempts] * workers)
```

该函数负责在可用工作者之间分发计算负载，同时收集计算结果。它以异步方式运行 evaluate_points_in_circle，并发执行多个对 evaluate_points_in_circle 的调用。

使用 SCOOP 处理映射函数

在处理列表或其他数据序列时，有一个非常有用的常见任务，即对列表中的每个元素应用相同的操作，然后收集结果。例如，列表更新操作可以这样从 Python 自带的 IDLE 中执行：

```
>>> items = [1,2,3,4,5,6,7,8,9,10]
>>> updated_items = []
>>> for x in items:
>>>    updated_items.append(x*2)

>>> updated_items
>>> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

这是一个常见操作。不过，Python 中已有一个内置特性，可以完成大部分工作。

Python 函数 map(aFunction, aSequence) 将传入的函数应用到可迭代对象中的每一项，并返回一个包含所有函数调用结果的列表。现在，同样的例子可以这样实现：

```
>>> items = [1,2,3,4,5,6,7,8,9,10]
>>> def multiplyFor2(x):return x*2
>>> print(list(map(multiplyFor2,items)))
>>> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

这里，我们向用户自定义的函数 multiplyFor2 中传入了 map 函数。它被应用到 items 列表中的每一项，最后在一个新列表中收集结果并打印。

另外，也可以将一个 lambda 函数（定义时未被绑定到标识符的函数）作为参数传入，不是只能传入普通函数。同样的例子现在可以这样实现：

```
>>> items = [1,2,3,4,5,6,7,8,9,10]
```

```
>>> print(list(map(lambda x:x*2,items)))
>>> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

内置函数 map 还具备性能优势，因为它比手写的 for 循环更快。

准备工作

SCOOP 模块定义了多个映射函数，支持可传播至其工作者的异步计算。这些函数如下所示。

- futures.map(func, iterables, kargs)：返回一个以输入中相同顺序迭代结果的生成器。可作为标准库中 map() 函数的并行版。
- futures.map_as_completed(func, iterables, kargs)：只要结果一可用即生成结果。
- futures.scoop.futures.mapReduce(mapFunc, reductionOp, iterables, kargs)：支持在应用 map() 函数后将减项函数（reduction function）并行化，返回单个元素。

具体实现

在此例中，我们将对比 MapReduce 函数的 SCOOP 实现和串行实现（serial implementation）：

```
"""
比较 SCOOP 版的 MapReduce 和串行实现
"""
import operator
import time

from scoop import futures


def simulateWorkload(inputData):
    time.sleep(0.01)
    return sum(inputData)


def CompareMapReduce():
    mapScoopTime = time.time()
    res = futures.mapReduce(
        simulateWorkload,
        operator.add,
        list([a] * a for a in range(1000)),
    )
    mapScoopTime = time.time() - mapScoopTime
    print("futures.map in SCOOP executed in {0:.3f}s \
           with result:{1}".format(
        mapScoopTime,
        res
    )
```

```
    )

    mapPythonTime = time.time()
    res = sum(
        map(
            simulateWorkload,
            list([a] * a for a in range(1000))
        )
    )
    mapPythonTime = time.time() - mapPythonTime
    print("map Python executed in: {0:.3f}s \
          with result: {1}".format(
        mapPythonTime,
        res
        )
    )
if __name__ == '__main__':
    CompareMapReduce()
```

如要运行该脚本，必须输入以下命令：

```
python -m scoop map_reduce.py
```

```
> [2015-06-12 20:13:25,602] launcher  INFO    SCOOP 0.7.2 dev on win32
using Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)], API: 1013
[2015-06-12 20:13:25,602] launcher  INFO Deploying 2 worker(s) over 1
host(s).
[2015-06-12 20:13:25,602] launcher  INFO Worker d--istribution:
[2015-06-12 20:13:25,602] launcher  INFO 127.0.0.1:       1 + origin
Launching 2 worker(s) using an unknown shell.
futures.map in SCOOP executed in 8.459s with result: 332833500
map Python executed in: 10.034s with result: 332833500
[2015-06-12 20:13:45,344] launcher  (127.0.0.1:2559) INFO    Root process is
done.
[2015-06-12 20:13:45,368] launcher  (127.0.0.1:2559) INFO    Finished
cleaning spawned subprocesses.
```

实例精解

在此例中，我们对比了 MapReduce 函数的 SCOOP 实现和串行实现。该脚本的核心是 CompareMapReduce() 函数，其中包含了两种实现。同样在该函数中，我们根据以下范式计算执行时间：

```
mapScoopTime = tme.time()
        # 运行 SCOOP 版的 MapReduce
mapScoopTime = time.time() - mapScoopTime

mapPythonTime = time.time()
        # 运行串行版的 MapReduce
mapPythonTime = time.time() - mapPythonTime
```

然后我们在输出中打印最终的执行时间：

```
futures.map in SCOOP executed in 8.459s with result: 332833500
map Python executed in: 10.034s with result: 332833500
```

要想获得可比较的执行时间，我们模拟真实的计算工作负载，在 simulatedWordload 函数中引入一个 time.sleep 语句：

```
def simulateWorkload(inputData, chose=None):
    time.sleep(0.01)
    return sum(inputData)
```

mapReduce 的 SCOOP 实现如下：

```
res = futures.mapReduce(
       simulateWorkload,
       operator.add,
       list([a] * a for a in range(1000)),
   )
```

futures.mapReduce 函数有以下参数。

- simulateWork：调用该函数可执行 Futures。还需记住的是，可调用对象必须返回一个值。
- operator.add：调用该函数可归约（reduce）Futures 运行结果。但是，它还必须接受两个参数，并返回一个单一值。
- list(.....)：这是将作为单独的 Future 传入可调用对象的可迭代对象。

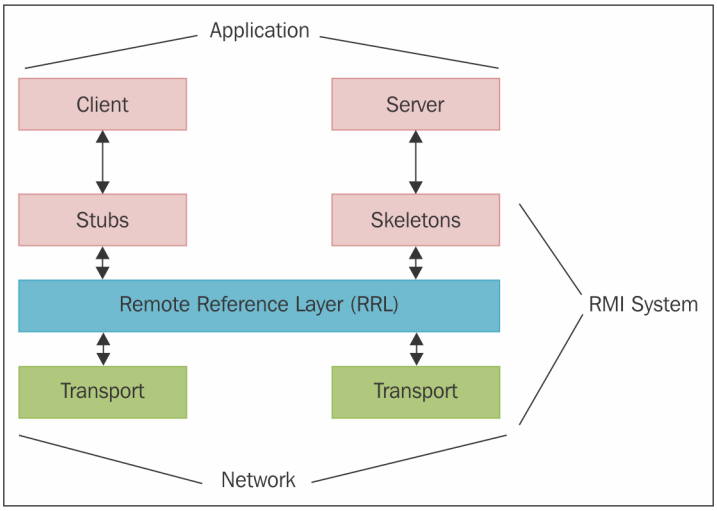
mapReduce 的串行实现如下：

```
res = sum(
    map(
        imulateWorkload,
        list([a] * a for a in range(1000))
    )
)
```

Python 中的标准 map() 函数中有两个参数：simulateWorkload 函数和可迭代对象 list()。不过，为了将结果归约，我们使用了 sum 函数。

使用 Pyro4 远程调用方法

Python 远程对象（Python Remote Objects，简称 Pyro4）是一个类似 Java 中远程方法调用（Remote Method Invocation，简称 RMI）的库，支持调用远程对象（属于不同的进程，可能位于另一台机器上）的方法，就好像它就是本地对象（同属于运行调用的那个进程），如下图所示。在这种意义上，远程方法调用技术可以从概念角度进行回溯。远程过程调用（remote procedure call，简称 RPC）的概念被按照面向对象范式重新组织（在面向对象的范式中，方法取代了过程）。在面向对象系统中使用远程方法调用机制，可以为项目带来极大的一致性、对称性优势，因为它可以让我们使用同一个概念工具来模拟分布式进程之间的交互，该工具也被用于表示一个应用或方法调用中不同对象的交互。



远程方法调用

从上图可以看出，Pyro4 支持以客户端 - 服务器的形式管理和分发对象。这意味着一个 Pyro4 系统的主要部分可以从一个调用远程对象的客户端切换到一个调用来提供函数的对象。值得注意的是，在远程调用期间，一直存在两个明显的部分，即客户端发起调用，服务器接收并执行客户端调用。最后，这个机制的管理均由 Pyro4 以分布式的方式进行。

准备工作

使用 `pip` 可以轻松安装 Pyro4 库；在命令行中输入：`pip install pyro` 即可。

也可以从 `https://github.com/irmen/ Pyro4` 处下载完整的包，然后从包目录下输入 Python 的 `setup.py` 安装命令进行安装。

在接下来的示例中，我们将使用 Windows 机器上的 Python 3.3 发行版。

具体实现

在本例中，我们将介绍如何使用 Pyro4 中间件构建并使用一个简单的客户端 - 服务器通信系统。因此，我们必须编写两个 Python 脚本。

服务器（server.py）的代码如下：

```
import Pyro4

class Server(object):
    def welcomeMessage(self, name):
        return ("Hi welcome " + str(name))

    def startServer():
        server = Server()
        daemon = Pyro4.Daemon()
        ns = Pyro4.locateNS()
        uri = daemon.register(server)
        ns.register("server", uri)
        print("Ready. Object uri =", uri)
        daemon.requestLoop()

if __name__ == "__main__":
    startServer()
```

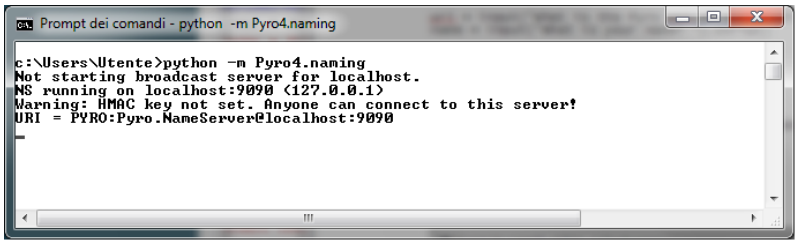
客户端（client.py）的代码如下：

```
import Pyro4

uri = input("What is the Pyro uri of the greeting object? ").strip()
name = input("What is your name? ").strip()
server = Pyro4.Proxy("PYRONAME:server")
print(server.welcomeMessage(name))
```

在运行示例之前，需要先运行一个 Pyro 名称服务器（name server）。为此，可以在命令行输入以下命令：

```
python -m Pyro4.naming
```

之后，会看到如下图所示的信息。



这意味着名称服务器已在你的网络中运行。然后，可以在两个独立的控制台窗口中分别启动服务器和客户端脚本。如要运行服务器，只需输入以下命令：

```
python server.py
```

现在，会看到类似下图所示的内容。

如要运行客户端，只需键入：

```
python client.py
```

之后，会出现类似下面的消息：

```
insert the PYRO4 server URI (help : PYRONAME:server)
```

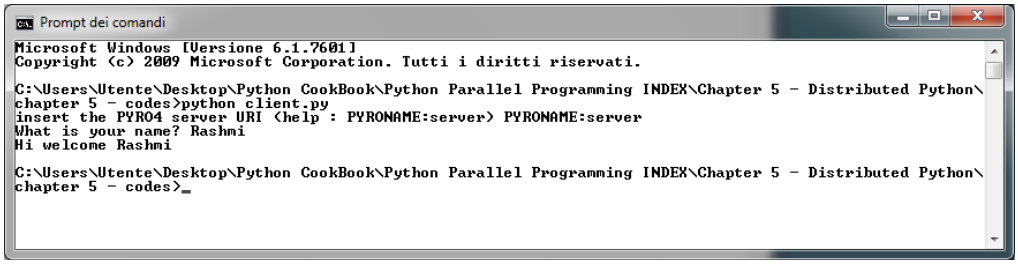
这里必须填写 Pyro4 服务器的名称，即 `PYRONAME:server`：

```
insert the PYRO4 server URI (help : PYRONAME:server) PYRONAME:server
```

然后会看到要求你输入自己名字的消息：

```
What is your name? Rashmi
```

最后，你将看到一则欢迎消息：`Hi welcome Rashmi`，如下图所示。



实例精解

服务器中含有可以远程访问的对象（Server 类）。在我们的示例中，该对象只有一个 welcomeMessage() 方法，返回一个带有客户端会话中插入的名称的字符串：

```
class Server(object):
    def welcomeMessage(self, name):
        return ("Hi welcome " + str (name))
```

如要启动服务器（startServer() 函数），必须按如下步骤进行操作：

1. 构建 Server 类的实例（名为 server）：server = Server()。
2. 创建一个 Pyro 守护程序（daemon）：daemon = Pyro4.Daemon()。Pyro4 使用守护程序对象将传入的调用请求分配给相应的对象。服务器必须创建一个负责管理实例的守护程序。每个服务器都有一个这样的守护程序，它掌握了该服务器提供的所有 Pyro 对象。
3. 执行该脚本之前，必须运行一个 Pyro 名称服务器。因此，我们要定位该名称服务器：ns = Pyro4.locateNS()。
4. 然后，需要将该服务器注册为 Pyro 对象 object。只有在 Pyro 守护程序的内部才知道该对象：uri = daemon.register(server)。它返回注册对象的 URI。
5. 最后，可以在名称服务器中使用一个名称注册该对象服务器：ns.register("server", uri)。
6. 函数运行结束时，将调用守护程序的 eventloop 方法。这将启动服务器的事件循环，并等待其他调用传入。

Pyro4 API 可以让开发者直观地分发对象。客户端脚本向服务器程序发送执行 welcome-Message() 方法的请求。该远程调用首先会创建一个代理对象（Proxy object）。事实上，Pyro4 客户端使用代理对象转发对远程对象的方法调用，并将结果传递回调用代码：

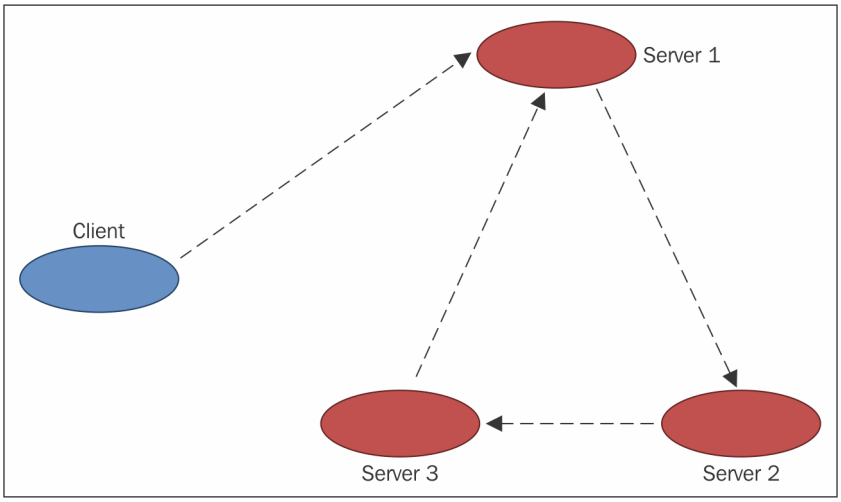
```
server = Pyro4.Proxy("PYRONAME:server")
```

现在，调用服务器上能够打印欢迎消息的方法：

```
print(server.welcomeMessage(name))
```

使用 Pyro4 链接对象

在该示例中，我们将介绍如何使用 Pyro4 创建一个对象链，其中的对象相互调用。假设我们想构建一个类似下图所示的分布式架构。



使用 Pyro4 链接对象

有 4 个对象：1 个客户端，以及按链式拓扑分布的 3 个服务器，如上图所示。客户端将请求转发至 Server1 并启动了链式调用（chain call），后者再将请求转发至 Server2。然后，它再调用链中的下一个对象 Server3。当 Server3 再次调用 Server1 时，链式调用结束。

我们即将介绍的这个示例，会特别关注如何管理远程对象。可以轻松地扩展这些对象，以支持更加复杂的分布式架构。

具体实现

如要使用 Pyro4 实现一个对象链，需要 5 个 Python 脚本。第一个是客户端（`test.py`）。以下是它的代码：

```
from __future__ import print_function
import Pyro4

obj = Pyro4.core.Proxy("PYRONAME:example.chain.A")
print("Result=%s" % obj.process(["hello"]))
```

每个服务器中都有一个明显的参数 `this` 以及一个 `next` 参数，前者在对象链中表示自身，后者定义了链中的下一个服务器（也就是 `this` 后面的那个服务器）。

如想了解对象链实现后的情况，可以查看与该示例有关的那张图。

- `server_1.py`：

```
from __future__ import print_function
import Pyro4
import chainTopology
```

```
this = "1"
next = "2"

servername = "example.chainTopology." + this

daemon = Pyro4.core.Daemon()
obj = chainTopology.Chain(this, next)
uri = daemon.register(obj)
ns = Pyro4.naming.locateNS()
ns.register(servername, uri)

# 进入服务循环
print("server_%s started " % this)
daemon.requestLoop()
```

- server_2.py：

```
from __future__ import print_function
import Pyro4
import chainTopology

this = "2"
next = "3"

servername = "example.chainTopology." + this

daemon = Pyro4.core.Daemon()
obj = chain.chainTopology(this, next)
uri = daemon.register(obj)
ns = Pyro4.naming.locateNS()
ns.register(servername, uri)

# 进入服务循环
print("server_%s started " % this)
daemon.requestLoop()
```

- server_3.py：

```
from __future__ import print_function
import Pyro4
import chainTopology

this = "3"
next = "1"
```

```
servername = "example.chainTopology." + this

daemon = Pyro4.core.Daemon()
obj = chain.chainTopology(this, next)
uri = daemon.register(obj)
ns = Pyro4.naming.locateNS()
ns.register(servername, uri)

# 进入服务循环
print("server_%s started " % this)
daemon.requestLoop()
```

最后一个脚本是 chain 对象，代码如下所示：

- chainTopology.py：

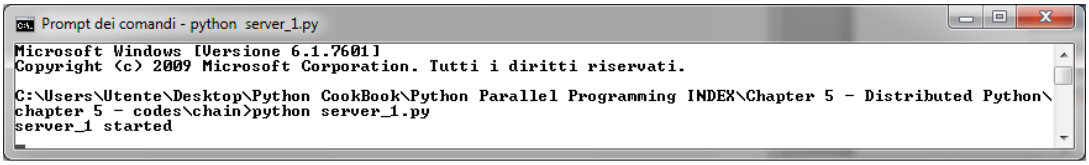
```
from __future__ import print_function
import Pyro4

class Chain(object):
    def __init__(self, name, next):
        self.name = name
        self.nextName = next
        self.next = None

    def process(self, message):
        if self.next is None:
            self.next = Pyro4.core.Proxy(
                "PYRONAME:example.chain." + self.nextName)
        if self.name in message:
            print("Back at %s; the chain is closed!" % self.name)
            return ["complete at " + self.name]
        else:
            print("%s forwarding the message to the object %s"
                  % (self.name, self.nextName))
            message.append(self.name)
            result = self.next.process(message)
            result.insert(0, "passed on from " + self.name)
            return result
```

如需执行该示例，首先要运行 Pyro4 名称服务器：

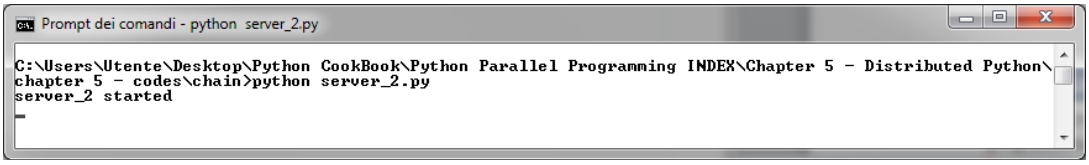
```
C:>python -m Pyro4.naming
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@localhost:9090
```

然后，运行三个服务器。打开三个命令提示符，然后输入 python server_name.py 命令。

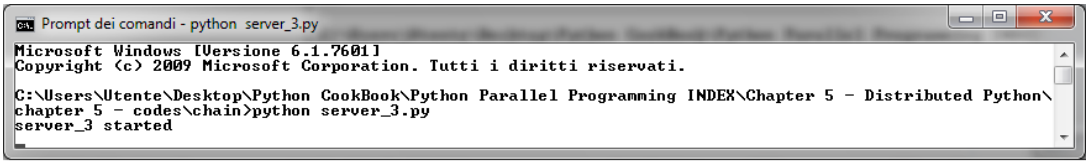
运行 server_1 之后，应该会出现类似下图所示的消息。



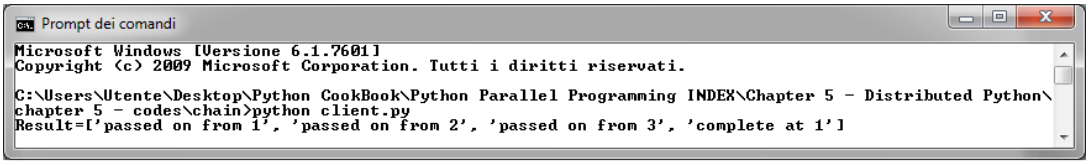
运行 server_2 之后，应该会出现类似下图所示的消息。



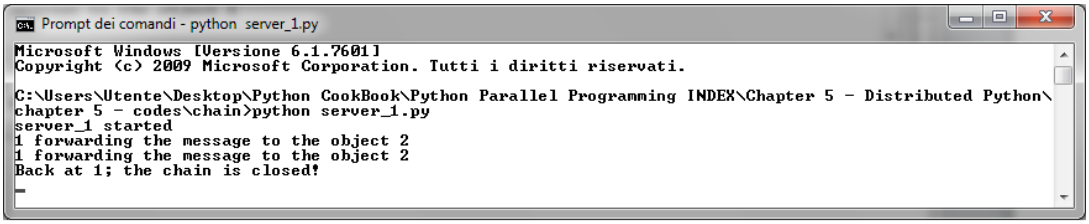
运行 server_3 之后，应该会出现类似下图所示的消息。



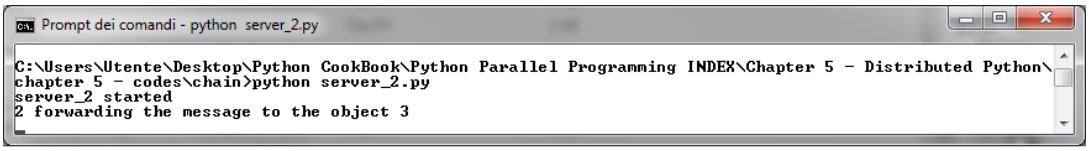
最后，必须从另一个命令行中运行 client.py 脚本。



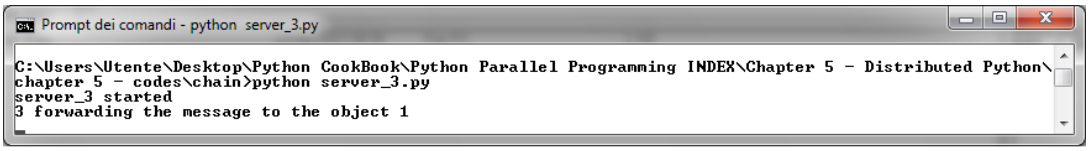
上面的消息显示，转发的请求经过了三个服务器的传递，当回到 server_1 时，任务完成。另外，我们这里重点关注当请求转发到链中的下一个对象时，对象服务器的行为。如想了解接下来会发生什么，请查看以下对 server_1 的截图中的消息：



server_2 的结果如下图所示。



server_3 的结果如下图所示。



实例精解

该示例的核心是定义在 chainTopology.py 中的 Chain 类。它支持三个服务器之间进行通信。实际上，每个服务器均调用该类，确定链中的下一个元素是什么（参看 chainTopology.py 中的方法进程）。另外，它使用 Pyro4.core.proxy 语句执行调用：

```
if self.next is None:
    self.next = Pyro4.core.Proxy(
        "PYRONAME:example.chain." + self.nextName)
```

如果链关闭（完成从 server_3 到 server_1 的最后一次调用），会打印关闭消息：

```
if self.name in message:
    print("Back at %s; the chain is closed!" % self.name)
    return ["complete at " + self.name]
```

如果链中还存在下一个元素，会打印一条转发消息：

```
print("%s forwarding the message to the object %s"
      % (self.name, self.nextName))
      message.append(self.name)
      result = self.next.process(message)
      result.insert(0, "passed on from " + self.name)
      return result
```

每个服务器的代码相同，只是链的当前元素和下一个元素的定义略有不同。例如，下面是第一个服务器（server_1）的定义：

```
this = "1"
next = "2"
```

代码中的其他行均相同，定义了与链中下一个元素之间的通信过程：

```
servername = "example.chainTopology." + this
daemon = Pyro4.core.Daemon()
obj = chainTopology.Chain(this, next)
uri = daemon.register(obj)
ns = Pyro4.naming.locateNS()
ns.register(servername, uri)
daemon.requestLoop()
```

最后，在客户端脚本中，我们通过调用链中的第一个元素（server_1）启动进程：

```
obj = Pyro4.core.Proxy("PYRONAME:example.chainTopology.1")
```

使用 Pyro4 开发一个客户端–服务器应用

在该示例中，我们将介绍如何使用 Pyro4 开发一个简单的客户端 - 服务器（C/S）应用。这里介绍的应用并不完整，但它已经具备了能够成功完成并可以在后续完善的全部方法。

客户端 - 服务器应用指的是一种网络架构，其中客户端电脑或终端连接至服务器，使用某个特定服务，如与其他客户端一起分享硬件或软件资源，共用相同的底层协议架构。在示例体系中，服务器管理着一个在线购物站点，而客户端则管理在站点上注册并连接进行购物的客户。

具体实现

为了简洁起见，我们使用 3 个脚本。第一个脚本定义了对象 client，用于管理客户，第二个脚本定义了对象 shop，第三个脚本定义了对象 server。

服务器（server.py）的代码如下：

```
#
#    Shop 服务器
#

from __future__ import print_function
import Pyro4
import shop

ns = Pyro4.naming.locateNS()
daemon = Pyro4.core.Daemon()
uri = daemon.register(shop.Shop())
ns.register("example.shop.Shop", uri)
print(list(ns.list(prefix="example.shop.").keys()))
daemon.requestLoop()
```

客户端（client.py）的代码如下：

```
from __future__ import print_function
import sys
import Pyro4

# Shop 客户端
class client(object):
    def __init__(self, name, cash):
        self.name = name
        self.cash = cash
    def doShopping_deposit_cash(self, Shop):
        print("\n*** %s is doing shopping with %s:"
              % (self.name, Shop.name()))
        print("Log on")
        Shop.logOn(self.name)
        print("Deposit money %s" % self.cash)
        Shop.deposit(self.name, self.cash)
        print("balance=%.2f" % Shop.balance(self.name))
        print("Deposit money %s" % self.cash)
        Shop.deposit(self.name, 50)
        print("balance=%.2f" % Shop.balance(self.name))
        print("Log out")
        Shop.logOut(self.name)

    def doShopping_buying_a_book(self, Shop):
        print("\n*** %s is doing shopping with %s:"
              % (self.name, Shop.name()))
        print("Log on")
        Shop.logOn(self.name)
        print("Deposit money %s" % self.cash)
        Shop.deposit(self.name, self.cash)
        print("balance=%.2f" % Shop.balance(self.name))
        print("%s is buying a book for %s$"
              % (self.name, 37))
        Shop.buy(self.name, 37)
        print("Log out")
        Shop.logOut(self.name)

if __name__ == '__main__':
    ns = Pyro4.naming.locateNS()
    uri = ns.lookup("example.shop.Shop")
    print(uri)
    Shop = Pyro4.core.Proxy(uri)
    meeta = client('Meeta', 50)
    rashmi = client('Rashmi', 100)
```

```
    rashmi.doShopping_buying_a_book(Shop)
    meeta.doShopping_deposit_cash(Shop)
    print("")
    print("")
    print("")
    print("")

    print("The accounts in the %s:" % Shop.name())
    accounts = Shop.allAccounts()
    for name in accounts.keys():
        print("  %s : %.2f"
              % (name, accounts[name]))
```

对象 shop（shop.py）的代码如下：

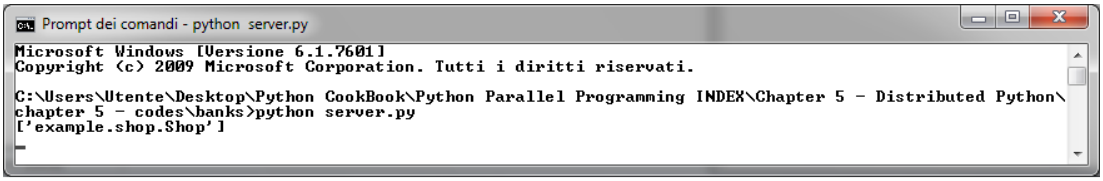
```
class Account(object):
    def __init__(self):
        self._balance = 0.0

    def pay(self, price):
        self._balance -= price

    def deposit(self, cash):
        self._balance += cash

    def balance(self):
        return self._balance


class Shop(object):
    def __init__(self):
        self.accounts = {}
        self.clients = ['Meeta', 'Rashmi', 'John', 'Ken']

    def name(self):
        return 'BuyAnythingOnline'

    def logOn(self, name):
        if name in self.clients:
            self.accounts[name] = Account()
        else:
            self.clients.append(name)
            self.accounts[name] = Account()

    def logOut(self, name):
        print('logout %s' % name)
```

```
    def deposit(self, name, amount):
        try:
            return self.accounts[name].deposit(amount)
        except KeyError:
            raise KeyError('unknown account')

    def balance(self, name):
        try:
            return self.accounts[name].balance()
        except KeyError:
            raise KeyError('unknown account')

    def allAccounts(self):
        accs = {}
        for name in self.accounts.keys():
            accs[name] = self.accounts[name].balance()
        return accs

    def buy(self, name, price):
        balance = self.accounts[name].balance()
        self.accounts[name].pay(price)
```

运行代码之前，必须首先启动 Pyro4 名称服务器：

```
C:>python -m Pyro4.naming
Not starting broadcast server for localhost.
NS running on localhost:9090 (127.0.0.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@localhost:9090
```

然后，使用 python server.py 命令启动服务器。运行命令之后，会出现类似下面截图中的情形。



最后，你应使用以下命令，模拟客户访问：

```
python client.py
```

运行上面的命令后，会打印如下文本：

```
C:\Users\Utente\Desktop\Python CookBook\Python Parallel Programming
INDEX\Chapter 5 - Distributed Python\

chapter 5 - codes\banks>python client.py
PYRO:obj_8c4a5b4ae7554c2c9feee5b0113902e0@localhost:59225

*** Rashmi is doing shopping with BuyAnythingOnline:
Log on
Deposit money 100
balance=100.00
Rashmi is buying a book for 37$
Log out

*** Meeta is doing shopping with BuyAnythingOnline:
Log on
Deposit money 50
balance=50.00
Deposit money 50
balance=100.00
Log out

The accounts in the BuyAnythingOnline:
  Meeta : 100.00
  Rashmi : 63.00
```

上面的输出显示了 Meeta 和 Rashmi 这两名客户之间的一次简单会话。

实例精解

应用的服务器端必须定位 Shop() 对象，调用如下语句即可：

```
ns = Pyro4.naming.locateNS()
```

然后，必须启用一个通信通道：

```
daemon = Pyro4.core.Daemon()
uri = daemon.register(shop.Shop())
ns.register("example.shop.Shop", uri)
daemon.requestLoop()
```

shop.py 脚本中包含了用于账户和店铺管理的类。shop 类管理每个账户，提供登入、登出的方法，管理客户的资金，并购买物品：

```
class Shop(object):

    def logOn(self, name):
```

```
        if name in self.clients :
            self.accounts[name] = Account()
        else :
            self.clients.append(name)
            self.accounts[name] = Account()

    def logOut(self, name):
        print('logout %s' %name)

    def deposit(self, name, amount):
       try:
            return self.accounts[name].deposit(amount)
        except KeyError:
            raise KeyError('unknown account')

    def balance(self, name):
        try:
            return self.accounts[name].balance()
        except KeyError:
            raise KeyError('unknown account')

    def buy(self, name, price):
        balance = self.accounts[name].balance()
        self.accounts[name].pay(price)
```

每个客户都有自己的 Account 对象，该对象提供了管理客户储蓄的方法：

```
class Account(object):
    def __init__(self):
        self._balance = 0.0

    def pay(self, price):
        self._balance -= price

    def deposit(self, cash):
        self._balance += cash

    def balance(self):
        return self._balance
```

最后，client.py 脚本中包含了用于启动客户模拟的类。在主程序中，我们实例化两个客户：Rashmi 和 Meeta。

```
meeta = client('Meeta',50)
rashmi = client('Rashmi',100)
rashmi.doShopping_buying_a_book(Shop)
```

```
meeta.doShopping_deposit_cash(Shop)
```

他们在购物网站上存储了一些钱，然后开始购物，如下所示：

- Rashmi 购买了一本书：

```
def doShopping_buying_a_book(self, Shop):
    Shop.logOn(self.name)
    Shop.deposit(self.name, self.cash)
    Shop.buy(self.name,37)
    Shop.logOut(self.name)
```

- Meeta 往账户中分两次存入 100 美元：

```
def doShopping_deposit_cash(self, Shop):
    Shop.logOn(self.name)
    Shop.deposit(self.name, self.cash)
    Shop.deposit(self.name, 50)
    Shop.logOut(self.name)
```

- 在模拟结束时，主程序报告 Meeta 和 Rashmi 账户中的余额：

```
print("The accounts in the %s:" % Shop.name())
    accounts = Shop.allAccounts()
    for name in accounts.keys():
        print("  %s : %.2f"
              % (name, accounts[name]))
```

使用PyCSP实现顺序进程通信

PyCSP 是基于通信的顺序进程（communicating sequential processes，简称 CSP）的一个 Python 模块，它是通过消息传递方式构建并发程序的一种编程范式。PyCSP 模块有以下特点：

- 进程间的消息交换。
- 通过线程使用共享内存。
- 通过通道完成消息交换。

通道支持：

- 进程间交换值。
- 进程同步。

PyCSP 允许使用不同类型的通道：One2One、One2Any、Any2One 和 Any2One。这些名称说明了可以通过通道进行通信的 writer 和 reader 的个数。

准备工作

可以使用如下命令，通过 pip 安装器安装 PyCSP：

```
pip install python-csp
```

另外，也可从 GitHub（https://github.com/futurecore/python-csp）下载完整的发行版。

下载并从安装目录下输入以下命令：

```
python setup.py install
```

在下面的示例中，我们将使用 Python 2.7。

实例精解

在第一个例子中，我们将介绍一下 PyCSP 的基本概念、进程和通道。因此，我们定义了名为 counter 和 printer 的两个进程，现在来看看如何定义这两个进程之间的通信。

请看以下代码：

```
from pycsp.parallel import *

@process
def processCounter(cout, limit):
    for i in xrange(limit):
        cout(i)
    poison(cout)

@process
def processPrinter(cin):
    while True:
        print cin(),

A = Channel('A')
Parallel(
    processCounter(A.writer(), limit=5),
    processPrinter(A.reader())
)

shutdown()
```

如果想执行上面的代码，可以在 Python 2.7 自带的 IDLE 下打开并按下运行按钮。之后会出现类似下面的输出：

```
Python 2.7.9 (default, Dec 10 2014, 12:28:03) [MSC v.1500 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
```

```
>>> ========================RESTART ============================
>>>
0123 4
>>>
```

具体实现

在此例中，我们使用了定义在 pycsp.parallel 模块中的函数：

```
from pycsp.parallel import *
```

该模块中有一个 Any2Any 通道类型，支持多个进程（依附在通道的两端）通过它进行通信。我们使用以下语句创建通道 A：

```
A = Channel('A')
```

这个新通道自动托管在当前的 Python 解释器中。对于每个导入了 pycsp.parallel 模块的 Python 解释器，只会列出负责处理 Python 解释器中启动的所有通道的端口。不过，该模块并没有为通道提供可用的名称服务器。因此，如果要连接到一个托管的通道，你必须知道其正确位置。

例如，如需连接到本机端口 8888 上的通道 B，我们输入以下代码：

```
A = pycsp.Channel('B', connect=('localhost', 8888))
```

PyCSP 中有三种管理通道的基本方法。

- channel.Disconnect()：允许退出 Python 解释器。其用在某个客户端 - 服务器设置里，其中客户端在收到服务器的回复后，会断开与服务器的连接。
- channel.reader()：创建并返回通道的 reader 端。
- channel.writer()：创建并返回通道的 writer 端。

我们使用 @process 装饰器表示一个进程。在 PyCSP 中，每个生成的 CSP 进程是以单个 OS 线程的形式实现的。在这个例子中，有两个进程：counter 和 printer。counter 进程有两个参数：cout 用于重定向输出，limit 定义要打印的项数：

```
@process
def processCounter(cout, limit):
    for i in xrange(limit):
        cout(i)
    poison(cout)
```

poison 语句 poison(count) 意味着通道的这一端被污染了。也就是说，之后所有这个通道上的读写操作都将抛出一个可以用于结束当前过程或断开通道连接的异常。还要注意，如果存在多个并发过程，这可能会导致竞争条件（race condition）。

printer 进程只有一个参数，即要打印的项，定义为 cin 变量：

```
@process
def processPrinter(cin):
    while True:
        print cin(),
```

这个脚本的核心是下面这行代码：

```
A = Channel('A')
```

它定义了 A 通道，允许两个进程之间进行通信。

最后，Parallel 语句如下所示：

```
Parallel(
    processCounter(A.writer(), limit=5),
    processPrinter(A.reader())
)
```

这会启动所有进程，只有在 counter 和 process 进程结束相互之间的通信时才会阻塞。这个语句代表了 CSP 的一个基本思想：并行进程通过利用 A 通道同步 I/O 来实现同步。一种实现方法是只在 counter 进程表示准备好向 printer 进程输出、printer 进程表示准备好接收来自 counter 进程的输入时，才允许进行 I/O 操作。如果二者有一个不成立，已做好准备的进程将进入等待队列，直到另一个进程做好准备。

每个 PyCSP 应用会创建一个服务器线程，用于管理通过通道传来的通信。因此，有必要在结束 PyCSP 应用时调用 shutdown() 方法：

```
shutdown()
```

PyCSP 提供了两个可以追踪执行过程的方法：

- TraceInit(<filename>, stdout=<True | False>)：用于开启追踪过程。
- TraceQuit()：用于停止追踪过程。

必须按以下方式使用这两个方法：

```
from pycsp.common.trace import *

TraceInit("trace.log")

"""
打算追踪的进程

"""
```

```
TraceQuit()
shutdown()
```

我们已经构建好了针对该示例的追踪记录（limit 参数等于 3）：

```
{'chan_name': 'A', 'type': 'Channel'}
{'chan_name': 'A', 'type': 'ChannelEndWrite'}
{'chan_name': 'A', 'type': 'ChannelEndRead'}
{'processes': [{'func_name': 'processCounter', 'process_id':
'9cb4b3720ed111e5bb4c0024813d643d.processCounter'}, {'func_name':
'processPrinter', 'process_id': '9cb63a0f0ed111e5993a0024813d643d.
processPrinter'}], 'process_id': '9c42428f0ed111e59ba10024813d643d.__
INIT__', 'type': 'BlockOnParallel'}
{'func_name': 'processCounter', 'process_id':
'9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'type':
'StartProcess'}
{'func_name': 'processPrinter', 'process_id':
'9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'type':
'StartProcess'}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'BlockOnWrite', 'id': 0}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'BlockOnRead', 'id': 0}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'DoneRead', 'id': 0}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'DoneWrite', 'id': 0}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'BlockOnWrite', 'id': 1}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'BlockOnRead', 'id': 1}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'DoneRead', 'id': 1}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'DoneWrite', 'id': 1}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'BlockOnWrite', 'id': 2}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'BlockOnRead', 'id': 2}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'DoneRead', 'id': 2}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'DoneWrite', 'id': 2}
{'process_id': '9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'chan_
name': 'A', 'type': 'Poison', 'id': 3}
{'func_name': 'processCounter', 'process_id':
```

```
'9cb4b3720ed111e5bb4c0024813d643d.processCounter', 'type': 'QuitProcess'}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'BlockOnRead', 'id': 3}
{'func_name': 'processPrinter', 'process_id':
'9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'type': 'QuitProcess'}
{'process_id': '9cb63a0f0ed111e5993a0024813d643d.processPrinter', 'chan_
name': 'A', 'type': 'Poison', 'id': 3}
{'processes': [{'func_name': 'processCounter', 'process_id':
'9cb4b3720ed111e5bb4c0024813d643d.processCounter'}, {'func_name':
'processPrinter', 'process_id': '9cb63a0f0ed111e5993a0024813d643d.
processPrinter'}], 'process_id': '9c42428f0ed111e59ba10024813d643d.__
INIT__', 'type': 'DoneParallel'}
{'type': 'TraceQuit'}
```

知识扩展

CSP 是一种用于描述并发进程间交互的形式语言。它属于竞争数学理论的范畴，叫作进程代数（algebra process）。在实践中，CSP 被用作一种规格书写工具，并用于验证多种系统的竞争特性。受 CSP 理论启发而编写的编程语言 Occam 现在被广泛用作并发编程语言。

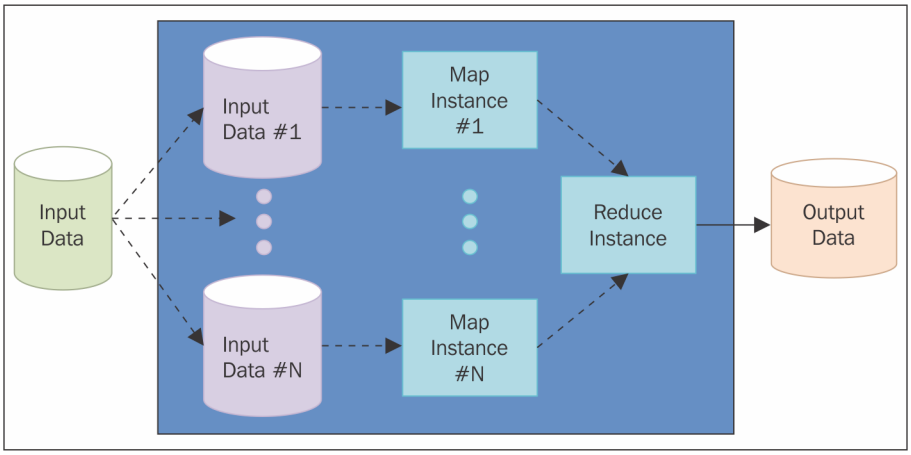
如对 CSP 理论感兴趣，建议阅读 Hoare 的原书，可从 http://www.usingcsp.com/cspbook.pdf 下载。

在Disco中使用 MapReduce

Disco 是一个基于谷歌推出的 MapReduce 框架的 Python 模块，支持在计算机集群中管理大规模分布式数据，如下图所示。使用 Disco 编写的应用可以在经济型计算机集群中运行，学习曲线非常短。事实上，与分布式进程相关的技术难点，如负载均衡、工作调度以及通信协议，可完全由 Disco 进行管理，不需开发者处理。

该模块的常见应用如下所示。

- Web 索引
- URL 访问计数
- 分布式排序



MapReduce 方案

Disco 中实现的 MapReduce 算法如下所示

- Map：主节点（master node）接受输入数据，将其拆分为更小的子任务，然后把工作分发给 slave 节点。单个 map 节点生成 `map()` 函数的中间结果，以 `[key, value]` 对的形式存储在一个分布式文件中。在这步的最后，主节点将获得该文件的地址。
- Reduce：主节点收集结果，并将 `[key, value]` 对中共享同一个键的值整合在一起，然后按照键进行排序（字母顺序或用户自定义）。`[key, IteratorList(value, value, ...)]` 这种形式的键值对被传递给运行归约函数 `reduce()` 的节点。

另外，存储在文件中的输出数据可以作为新的 `map` 和 `reduce` 过程的输入数据，这样可以将多个 MapReduce 工作拼接在一起。

准备工作

Disco 模块可在 `https://github.com/ Discoproject/Disco` 下载。

你需要使用 Linux/UNIX 系统才能安装 Disco。

以下是安装的前提条件（每个服务器上均如此）：

- SSH 守护程序和客户端。
- Erlang/OTP R14A 或更高版本。
- Python 2.6.6 或更高版本，Python 3.2 或更高版本。

最后，输入以下代码即可安装 Disco：

```
git clone git://github.com/Discoproject/Disco.git $Disco_HOME
cd $Disco_HOME
make
cd lib && python setup.py install --user && cd ..
bin/Disco nodaemon
```

下一步是在 Disco 集群的所有服务器中启用无须密码登录。如果是单一机器安装，必须运行如下命令：

```
ssh-keygen -N '' -f ~/.ssh/id_dsa
```

然后输入以下代码：

```
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

现在，如果你想登录集群或本地的所有服务器，在第一次成功登录之后，以后就不用再输入密码或回答问题了。

有关 Disco 安装的问题，请参考 http://Disco.readthedocs.org/en/latest/intro.html。

在接下来的示例中，我们将在一台 Linux 机器上使用 Python 2.7 发行版。

具体实现

在这个示例中，我们使用 Disco 模块来解决一个常见的 MapReduce 问题。给定一段文本，统计文本中某些词的出现次数：

```
from Disco.core import Job, result_iterator

    def map(line, params):
        import string
        for word in line.split():
            strippedWord = word.translate\
                (string.maketrans("", ""), string.punctuation)
            yield strippedWord, 1

    def reduce(iter, params):
        from Disco.util import kvgroup
        for word, counts in kvgroup(sorted(iter)):
            yield word, sum(counts)

if __name__ == '__main__':
    job = Job().run(input="There are known knowns.\
                           These are things we know that we know.\
```

```
                        There are known unknowns. \
                        That is to say,\
                        there are things that \
                        we know we do not know.\
                        But there are also unknown unknowns.\
                        There are things \
                        we do not know we do not know",
                    map=map,
                    reduce=reduce)

sort_in_numerical_order =\
                          open('SortNumerical.txt', 'w')
sort_in_alpbabetically_order = \
                               open('SortAlphabetical.txt', 'w')

wordCount = []
for word, count in \
    result_iterator(job.wait(show=True)):
    sort_in_alpbabetically_order.write('%s \t %d\n' %
                    (str(word), int(count)))
    wordCount.append((word, count))

sortedWordCount = sorted(wordCount,
                         key=lambda count: count[1],
                         reverse=True)

for word, count in sortedWordCount:
    sort_in_numerical_order.write('%s \t %d\n'
                                  % (str(word), int(count)))

sort_in_alpbabetically_order.close()
sort_in_numerical_order.close()
```

运行该脚本之后，会得到两个文件，已在下表中详细列出其内容。

Sortnumerical.txt		SortAlphabetical.txt	
6	are	also	1
6	know	are	6
6	we	but	1
5	there	do	3
3	do	is	1
3	not	know	6
3	that	known	2

续表

Sortnumerical.txt		SortAlphabetical.txt	
3	things	knowns	1
2	known	not	3
2	unknowns	say	1
1	also	to	1
1	but	that	3
1	is	there	5
1	knowns	these	1
1	say	things	3
1	to	unknown	1
1	these	unknowns	2
1	unknown	we	6

实例精解

该例的核心是 map 和 reduce 函数。Disco 中的 map 函数有两个参数，line 表示要分析的句子。不过，此例中将不考虑 params 参数。

这里，我们将句子分割成一个或多个单词，不考虑标点符号，并且所有单词都转换成小写字母：

```
def map(line, params):
    import string
    for word in line.split():
       strippedWord = word.translate\
                      (string.maketrans("","""), string.punctuation)
       yield strippedWord, 1
```

对一行文本执行 map 函数的结果是一系列元组，形式为键值对。例如，There are known knowns 这个句子将返回如下形式的结果：

```
[("There", 1), ("are", 1), ("known", 1), ("knowns",1)]
```

记住，MapReduce 框架可操作的数据集非常大，比单台机器上常见的内存要大很多，因此 map 函数最后的 yield 关键字将让 Disco 以更加聪明的方式管理数据集。reduce 函数接受两个参数：iter 指的是可迭代对象（行为类似列表数据结构），而 map 函数中也有的 params 参数在这里还是先忽略。

使用 Python 函数将每个可迭代对象按照字母顺序进行排序：

```
def reduce(iter, params):
```

```
    from Disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
       yield word, sum(counts)
```

我们在排好序的列表上应用 Disco 中的 kvgroup 函数，依次将等同键的值归组在一起。最后，通过 Python 函数 sum 计算出文本中每个单词的出现次数。

在脚本的主要部分，我们使用 Disco 的 job 函数来执行 mapReduce 函数：

```
job = Job().run(input="There are known knowns.\
                       These are things we know that we know.\
                       There are known unknowns. \
                       That is to say,\
                       there are things that \
                       we know we do not know.\
                       But there are also unknown unknowns.\
                       There are things \
                       we do not know we do not know",
                map=map,
                reduce=reduce)
```

最后，将结果按照数字大小和字母顺序进行排序，并打印到两个输出文件中：

```
sort_in_numerical_order = open('SortNumerical.txt', 'w')
sort_in_alpbabetically_order = open('SortAlphabetical.txt', 'w')
```

知识扩展

Disco 是一个非常强大的框架，具备许多不同的功能。但是本书不会对该模块进行完整的介绍。

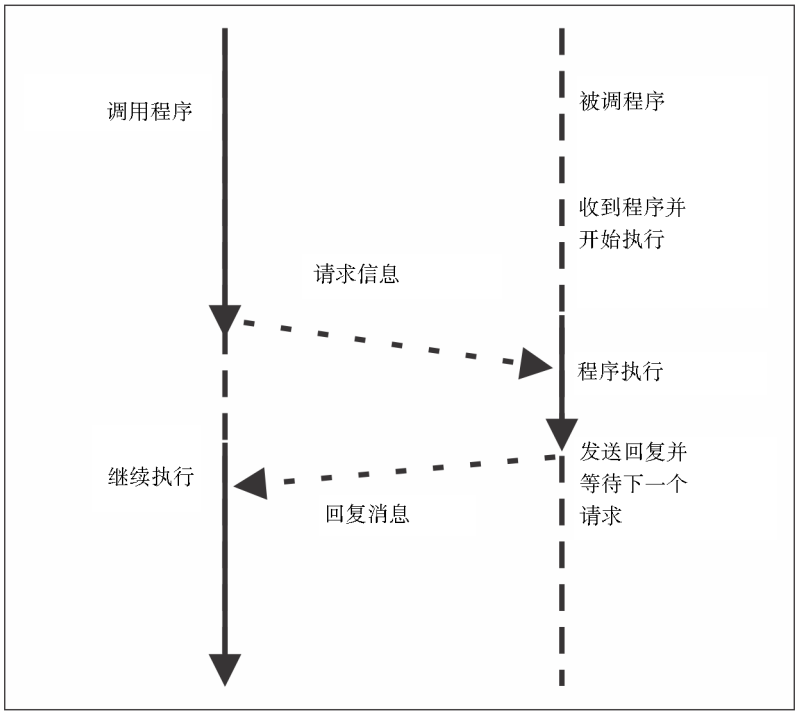
如想完整了解，请查看 http://Discoproject.org/。

使用 RPyC 调用远程过程

Remote Python Call（RPyC）是一个用于远程过程调用和分布式计算的 Python 模块。RPC 的基础理念是，提供将控制权从一个程序（客户端）转交给另一个程序（服务器）的机制，类似于集中式程序中子协程（subroutine）的调用过程，如下图所示。这种方法的好处在于，它的语义和语法非常简单，而且类似集中式的函数调用。在调用过程时，客户端进程暂停，直到服务器进程执行完必要的计算，并返回计算的结果。该方法之所以有效，是因为客户端 - 服务器通信以过程调用的形式进行，而不是调用传输层，因此通过将网络操作放在被称为 stub 的

本地过程中，可以把全部细节隐藏起来，不需应用程序直接操作。RPyC 的主要特性如下：

- 在语法透明性上，远程过程调用的语法与本地调用语法相同。
- 在语义透明性上，远程过程调用在语义上等同于本地调用。
- 支持同步和异步通信。
- 对称通信协议意味着客户端和服务器均可服务请求。



远程过程调用模式

准备工作

使用 pip 安装器，安装过程非常简单。在命令行中输入以下命令：

```
pip install rpyc
```

你也可以前往 https://github.com/tomerfiliba/rpyc 下载完整的包（.zip 文件）。最后，从包目录运行如下命令即可安装 rpyc：Python setup.py。

安装完成之后，你可以自行摸索如何使用该库。在我们的示例中，将在同一台机器（即本地 localhost）中运行客户端和服务器。使用 rpyc 运行服务器很简单：前往 rpyc 包目录中的 ../rpyc-master/bin 目录，然后执行 rpyc_classic.py 文件：

```
C:\ Python CookBook\ Chapter 5- Distributed Python\rpyc-master\bin>python
rpyc_classic.py
```

运行该脚本后，会从命令行看到如下输出信息：

```
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
```

具体实现

我们现在可以开始第一个示例，介绍如何重定向远程过程的 stdout：

```
import rpyc
import sys
c = rpyc.classic.connect("localhost")
c.execute("print ('hi python cookbook')")
c.modules.sys.stdout = sys.stdout
c.execute("print ('hi here')")
```

运行该脚本后，会在服务器端看到被重定向的输出：

```
C:\Python CookBook\Chapter 5- Distributed Python\rpyc-master\bin>python
rpyc_classic.py
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
INFO:SLAVE/18812:accepted 127.0.0.1:6279
INFO:SLAVE/18812:welcome [127.0.0.1]:6279
hi python cookbook
```

实例精解

第一步是运行一个连接服务器的客户端：

```
import rpyc

c = rpyc.classic.connect("localhost")
```

这里，客户端语句 rpyc.classic.connect(host, port) 会创建一个与给定主机和端口的套接字连接。套接字定义了连接的端点。rpyc 使用套接字与可能分布于不同机器中的其他程序进行通信。

接下来，是如下语句：

```
c.execute("print ('hi python cookbook')")
```

这将在服务器上执行打印语句（一个远程 exec 语句）。

6
使用Python进行GPU编程

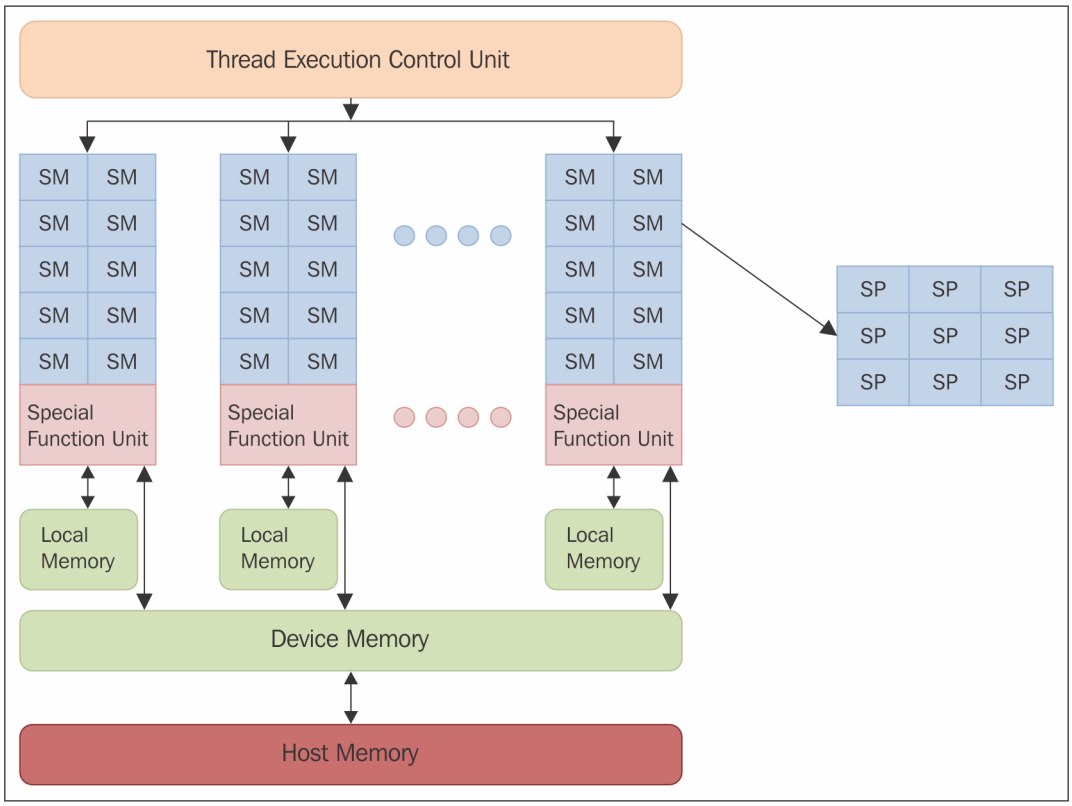
本章主要内容：

- 使用 PyCUDA 模块
- 如何构建一个 PyCUDA 应用
- 通过矩阵操作理解 PyCUDA 内存模型
- 使用 GPUArray 调用内核
- 使用 PyCUDA 对逐元素表达式求值
- 使用 PyCUDA 进行 MapReduce 操作
- 使用 NumbaPro 进行 GPU 编程
- 通过 NumbaPro 使用 GPU 加速的库
- 使用 PyOpenCL 模块
- 如何构建一个 PyOpenCL 应用
- 使用 PyOpenCL 对逐元素表达式求值
- 使用 PyOpenCL 测试 GPU 应用

介绍

图形处理器（graphics processing unit，简称 GPU）是一个专门用于处理数据以使用多边形基本体（polygonal primitives）渲染图像的电子线路板。尽管是被设计用来渲染图像的，但 GPU 一直在不断演变，变得越来越复杂，也越来越高效，能满足实时渲染和离线渲染需求，可以高效地执行科学计算。GPU 的特色是具备高度并行化的结构，因此它能以高效的方式处理大规模数据集。这一特性与图形硬件性能的快速提升、高可编程性结合在一起，使得科学界开始关注 GPU，考虑其用于除渲染图像之外的其他用途。传统 GPU 是功能固定的设备，整个渲染管道都构建在硬件中。这限制了图形程序员，迫使他们使用不同的、高效及高质量的渲染算法。因此，后来开发出了一种新型 GPU，使用数百万轻量级的并行核心构建而成，可使用

着色器（shader）对其编程，实现图形渲染。这是计算机图形领域和游戏行业取得的最大进展之一。由于出现了大量可编程的核心，GPU 生产商开始研发用于并行编程的模型。每个 GPU 都由多个被称为流式多处理器（Streaming Multiprocessor，简称 SM）的处理单元组成，这些处理单元代表了并行的第一个逻辑层；而且，每个 SM 之间是相互独立且同时运转的，GPU 架构如下图所示。



GPU 架构

每个 SM 包含一组流式处理器（Stream Processor，简称 SP），每个流式处理器都具备一个真正的执行核心，可以线性地运行一个线程。SP 是执行逻辑中的最小单元，代表了更细粒度并行的层次。SM 和 SP 的区分在本质上是结构化的，不过可以进一步勾勒出 GPU 中 SP 的逻辑组织，这些 SP 聚集在逻辑区中，这些逻辑区都共享一种特定的执行模式。组成一组 SP 的所有核心同时执行相同的指令。这就是我们在本书第 1 章中介绍过的单指令多数据（Single instruction, multiple data，简称 SIMD）模式。

每个 SM 都有一些寄存器（register），寄存器是可以快速访问的内存区域，这块区域是临

时的，只能本地访问（不同核心之间无法共享），而且大小有限制。它可以用于存储单个核心中经常使用的值。图形处理器通用计算（general-purpose computing on graphics processing units，简称 GP-GPU）这个领域致力于研究利用 GPU 计算能力快速执行计算所需的技术，这也得益于 GPU 内部高度的并行化。如前所述，GPU 的结构与传统处理器有很大区别；因此，它们存在的问题也完全不同，需要使用特定的编程技巧。图形处理器最突出的特性就是拥有大量可用的处理核心，可以运行许多相互竞争的执行线程，这些线程将部分同步式地执行相同的操作。如果希望将工作分成许多小部分，针对不同的数据执行相同的操作，该特性将十分有用且高效。相反，如果操作非常注重线性化和逻辑顺序，就很难充分利用这一架构。GPU 计算的编程范式被称为流式处理（Stream Processing），因为数据可以被看作一个全部由数值组成的流，在这个流上同步执行了相同的操作。

当前，能够最有效地利用 GPU 计算能力的解决方案是软件库 CUDA 和 OpenCL。在接下来的示例中，我们将介绍这些软件库在 Python 编程语言中的实现。

使用 PyCUDA 模块

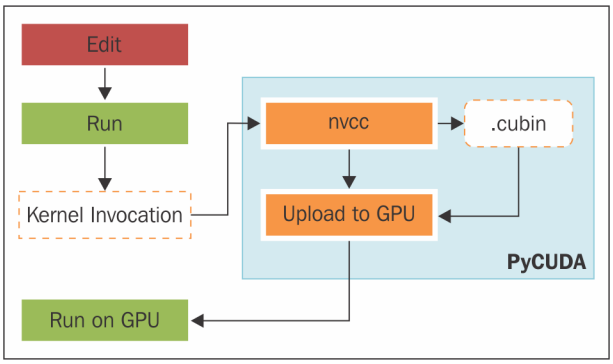
PyCUDA 是 NVIDIA 开发的 GPU 编程软件库 CUDA（Compute Unified Device Architecture）在 Python 中的封装。CUDA 编程模型是理解正确使用 PyCUDA 进行 GPU 编程的起点。如果想做到正确使用该工具，并且弄懂接下来的示例中介绍的内容，必须先理解和领会一些概念。

混合编程模型

CUDA 的“混合”编程模型是通过多个特定的 C 语言标准库扩展实现的，PyCUDA 亦然。只要有可能，就会创建这些扩展，语法上类似 C 标准库中的函数调用。这样，我们就能以相对简单的方式使用包含主机（host）和设备（device）代码的混合编程模型。NVCC 编译器对这两个逻辑部件进行管理。以下是对该编译器工作原理的简要描述：

1. 将设备代码与主机代码分离。
2. 调用默认编译器（例如，GCC）编译主机代码。
3. 以二进制（Cubin 对象）或汇编代码（PTX 代码）的形式构建设备代码。
4. 生成一个主机键“global”，其中也包含 PTX 代码。

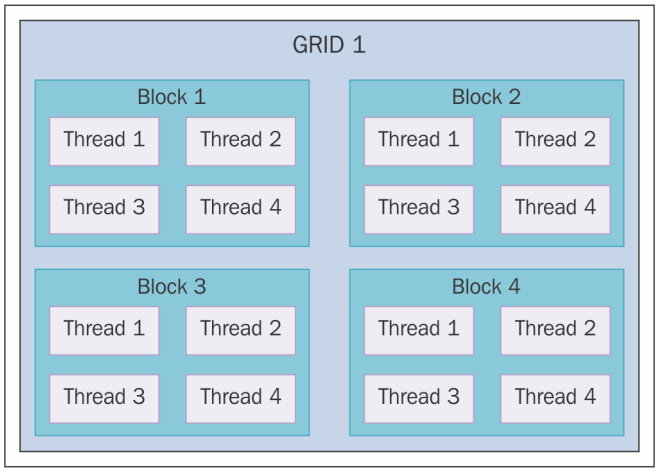
编译后的 CUDA 代码将在运行时由驱动器转换为特定于设备的二进制文件。上面提到的所有步骤都是 PyCUDA 在运行时完成的，因此它也算是一个即时（just-in-time，简称 JIT）编译器。这一方式的缺点是应用的加载时间会增加，因为这是唯一保持向“前”兼容的方法，即可以在实际编译时不存在的设备上执行运算。所以，JIT 编译可以让应用与构建于更高计算能力架构之上的未来设备兼容，因此它还不能生成任何二进制代码。PyCUDA 的执行模式如下图所示。



PyCUDA 执行模式

内核与线程层级

CUDA 程序的一个重要元素就是其内核（kernel）。它代表可以并行执行的代码，其基础规格说明将在稍后举例说明。每个内核的执行均由叫作线程（thread）的计算单元完成。与 CPU 中的线程不同，GPU 线程更加轻量，上下文的切换不会影响代码性能，因为切换可以说是瞬时完成的。为了确定运行一个内核所需的线程数及其逻辑组织形式，CUDA 定义了一个二层结构。在最高一层中，定义了所谓的区块网格（grid of blocks）。这个网格代表了线程区块所在的二维结构，而这些线程区块则是三维的，示意图如下所示。



PyCUDA 二层结构中（三维）线程的分布

基于这一结构，内核函数被调用时，必须提供额外的参数，来准确指定网格和区块大小。

准备工作

在维基百科页面 http://wiki.tiker.net/PyCuda/Installation 中，解释了在主要操作系统（Linux、Mac 和 Windows）中安装 PyCUDA 的基本步骤。

按照这些步骤，你可以针对 Python 2.7 发行版构建一个 32 位的 PyCUDA 库。

1. 第 1 步是下载并安装 NVIDIA 提供的用于 CUDA 开发的所有组件（参考 https://developer.nvidia.com/cuda-toolkit-archive 中的介绍），针对各个版本的组件都有，可参见下图。这些组件如下所示。
   - CUDA 工具包：http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/cudatoolkit_4.2.9_ win_32.msi。
   - NVIDIA GPU 计算 SDK：http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.9_win_32.exe。
   - NVIDIA CUDA 开发驱动器：http://developer.download.nvidia.com/compute/cuda/4_2/rel/drivers/devdriver_4.2_winvista-win7_32_301.32_general.exe。
2. 下载并安装 NumPy（针对 32 位 Python 2.7 的版本）和 Visual Studio C++ 2008 Express（记住，要设置全部系统变量）。
3. 打开位于 /Python27/lib/distutils/ 的文件 msvc9compiler.py。在第 64 行（ld_args.append ('/IMPLIB:' + implib_file)）之后，加上一行新代码：ld_args.append('/MANIFEST')。
4. 从 https://pypi.python.org/pypi/pycuda 下载 PyCUDA。
5. 打开 Visual Studio 2008 命令提示符：单击“开始”，前往“所有程序 | Microsoft Visual Studio 2008 | Visual Studio Tools | Visual Studio Command Prompt (2008)”，然后按以下步骤进行操作。

   （1）进入 PyCuda 目录。

   （2）执行 python configure.py 命令。

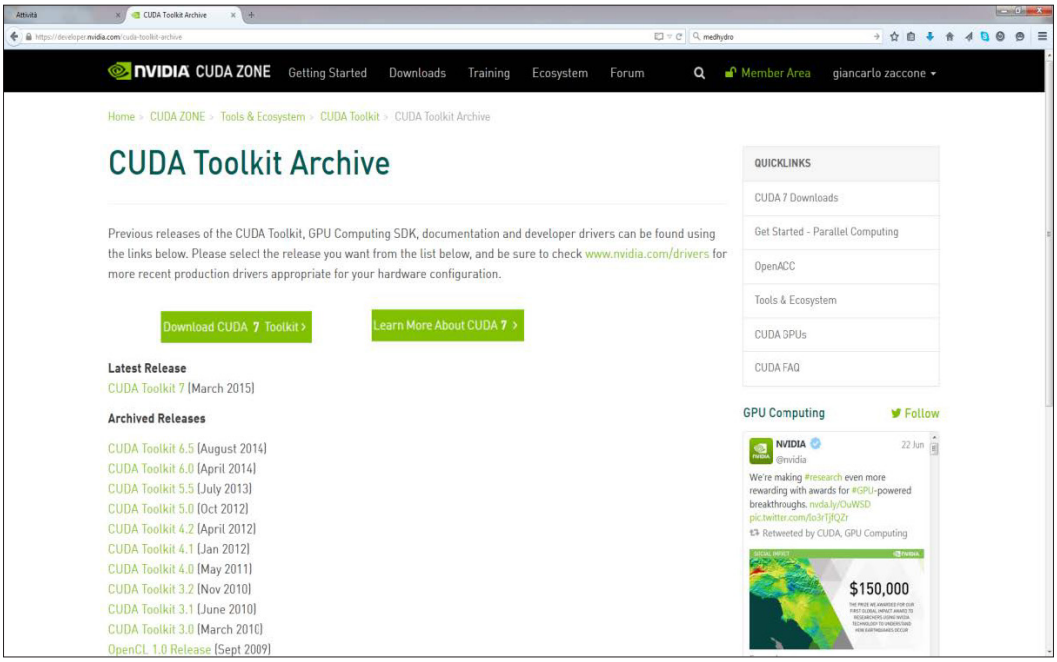
   （3）编辑创建的文件 siteconf.py：

```
BOOST_INC_DIR = []
BOOST_LIB_DIR = []
BOOST_COMPILER ='gcc43'
USE_SHIPPED_BOOST = True
BOOST_PYTHON_LIBNAME = ['boost_python']
BOOST_THREAD_LIBNAME = ['boost_thread']
CUDA_TRACE = False
CUDA_ROOT ='C:\\Program Files\\NVIDIA GPU Computing\\Toolkit\\
CUDA\\v4.2'
```

```
CUDA_ENABLE_GL = False
CUDA_ENABLE_CURAND = True
CUDADRV_LIB_DIR = ['${CUDA_ROOT}/lib/Win32']
CUDADRV_LIBNAME = ['cuda']
CUDART_LIB_DIR = ['${CUDA_ROOT}/lib/Win32']
CUDART_LIBNAME = ['cudart']
CURAND_LIB_DIR = ['${CUDA_ROOT}/lib/Win32']
CURAND_LIBNAME = ['curand']
CXXFLAGS = ['/EHsc']
LDFLAGS = ['/FORCE']
```

6. 最后，在 Visual Studio 2008 命令提示符中输入以下命令，安装 PyCUDA：

```
python setup.py build
python setup.py install
```



CUDA 工具包下载页面

具体实现

下面这个示例有两个功能。第一是验证 PyCUDA 是否正常安装，第二是读取并打印 GPU 显卡的特征信息：

```
import pycuda.driver as drv
drv.init()
```

```
print "%d device(s) found." % drv.Device.count()
for ordinal in range(drv.Device.count()):
    dev = drv.Device(ordinal)
    print "Device #%d: %s" % (ordinal, dev.name())
    print "Compute Capability: %d.%d" % dev.compute_capability()
    print "Total Memory: %s KB" % (dev.total_memory()//(1024))
```

运行上述代码之后，应该会看到类似下面的输出：

```
C:\ Python CookBook\ Chapter 6 - GPU Programming with Python\Chapter 6 -

codes>python PyCudaInstallation.py
1 device(s) found.
Device #0: GeForce GT 240
 Compute Capability: 1.2
 Total Memory: 1048576 KB
```

实例精解

上述代码的执行非常简单。在这段代码中，导入并初始化了 pycuda.driver：

```
import pycuda.driver as drv
drv.init()
```

pycuda.driver 暴露了 CUDA 的驱动器层编程接口，这比 CUDA C 语言中的“运行时层”编程接口更加灵活，而且拥有运行时所不具备的一些特性。

然后，程序遍历 drv.Device.count() 次，每发现一块 GPU 显卡，就打印显卡的名称和主要特征（计算能力及总内存）：

```
print "Device #%d: %s" % (ordinal, dev.name())
print "Compute Capability: %d.%d" % dev.compute_capability()
print "Total Memory: %s KB" % (dev.total_memory()//(1024))
```

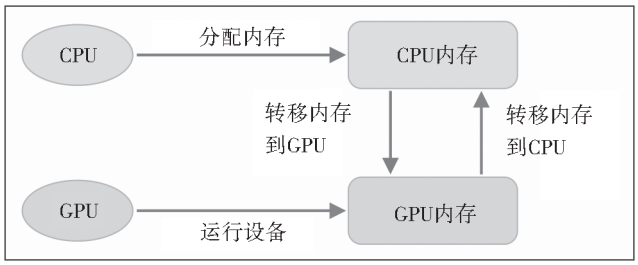
知识扩展

- PyCUDA 由 Andreas Klöckner（http://mathema.tician.de/aboutme/）开发。更多有关 PyCUDA 的信息，可参考 http://documen.tician.de/pycuda/。

如何构建一个 PyCUDA 应用

PyCUDA 编程模式是为了能同时在 CPU 和 GPU 上执行程序而设计的，其中线性部分在 CPU 上执行，更耗资源的数值计算部分则在 GPU 上执行。以线性模式执行的阶段是在 CPU（主机）上实现和执行的，而以并行模式执行的步骤则是在 GPU（设备）上实现并执行的。在设备上并行执行的函数被称为内核。在设备上执行通用函数内核的步骤如下所示，示意图如下图所示。

1. 第 1 步是在设备上分配内存。
2. 然后，需要将数据从主机内存转移到设备上分配的内存中。
3. 接下来，需要运行设备：
   （1）运行配置程序。
   （2）调用内核函数。
4. 然后，需要将结果从设备内存转移到主机内存中。
5. 最后，释放设备上分配的内存。



PyCUDA 编程模式

具体实现

为了更好地介绍 PyCUDA 工作流，我们来看一个 5×5 的随机数组，并参照以下步骤进行操作。

1. 在 CPU 上创建一个 5×5 的数组。
2. 将数组转移到 GPU。
3. 在 GPU 中 对数组执行一个操作（将数组中所有的项翻一倍）。
4. 将数组从 GPU 转移到 CPU。
5. 打印结果。

上述步骤的代码实现如下所示。

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule

import numpy

a = numpy.random.randn(5, 5)
a = a.astype(numpy.float32)
```

```
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)

mod = SourceModule(" ""
    __global__ void doubleMatrix(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
   " "")

func = mod.get_function("doubleMatrix")
func(a_gpu, block=(5, 5, 1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print("ORIGINAL MATRIX")
print a
print("DOUBLED MATRIX AFTER PyCUDA EXECUTION")
print a_doubled
```

示例的输出应该是类似这样的：

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\ >python
PyCudaWorkflow.py
ORIGINAL MATRIX
[[-0.59975582  1.93627465  0.65337795  0.13205571 -0.46468592]
 [ 0.01441949  1.40946579  0.5343408  -0.46614054 -0.31727529]
 [-0.06868593  1.21149373 -0.6035406  -1.29117763  0.47762445]
 [ 0.36176383 -1.443097    1.21592784 -1.04906416 -1.18935871]
 [-0.06960868 -1.44647694 -1.22041082  1.17092752  0.3686313 ]]
DOUBLED MATRIX AFTER PyCUDA EXECUTION
[[-1.19951165  3.8725493   1.3067559   0.26411143 -0.92937183]
 [ 0.02883899  2.81893158  1.0686816  -0.93228108 -0.63455057]
 [-0.13737187  2.42298746 -1.2070812  -2.58235526  0.95524889]
 [ 0.72352767 -1.443097    1.21592784 -1.04906416 -1.18935871]
 [-0.06960868 -1.44647694 -1.22041082  1.17092752  0.3686313 ]]
```

实例精解

上述代码的开头是以下导入语句：

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
```

import pycuda.autoinit 语句自动根据 GPU 可用性和数量选择要使用的 GPU。这也将

创建一个在接下来的代码运行中所需的 GPU 上下文。如果需要，选中的设备和创建的上下文均可从 pycuda.autoinit 中访问，并用作可导入的标识，而 SourceModule 组件则是一个必须编写 GPU 所需的类 C 代码的对象。

第一步是生成一个 5×5 的输入矩阵。由于大部分 GPU 计算都涉及大量的数据数组，因此必须导入 numpy 模块：

```
import numpy
a = numpy.random.randn(5, 5)
```

然后，矩阵中的项被转换为单精度模式，许多 NVIDIA 显卡只支持单精度：

```
a = a.astype(numpy.float32)
```

实现 GPU 所需完成的第一个操作是从主机内存(CPU)中将输入数组加载至设备中(GPU)。而且，必须在该操作一开始就完成，包括两个步骤，分别靠调用 PyCUDA 提供的两个函数实现：

- 设备上的内存分配通过 cuda.mem_alloc 完成。在执行函数内核时，设备和主机内存不能进行通信。这意味着，要在设备上并行运行一个函数，相关的数据必须已经存在于设备内存中。在将数据从主机内存复制至设备内存之前，必须分配设备上所需的内存：a_gpu = cuda.mem_alloc(a.nbytes)。
- 通过以下函数，将矩阵从主机内存复制至设备内存中：

  cuda.memcpy_htod(a_gpu, a)。

还要注意，a_gpu 是一维的，在设备上我们需要将其视为一维处理。所有这些操作都不要求调用内核，由主处理器直接完成。SourceModule 实体用于定义内核函数（类似 C 函数）doubleMatrix，该函数将每个数组项乘以 2：

```
mod = SourceModule(" ""
    __global__ void doubleMatrix(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
   " "")
```

__global__ 限定符表示，doubleMatrix 函数将在设备上执行。只有 CUDA 中的 NVCC 编译器将执行该任务。

我们来看看函数的主体：

```
int idx = threadIdx.x + threadIdx.y*4;
```

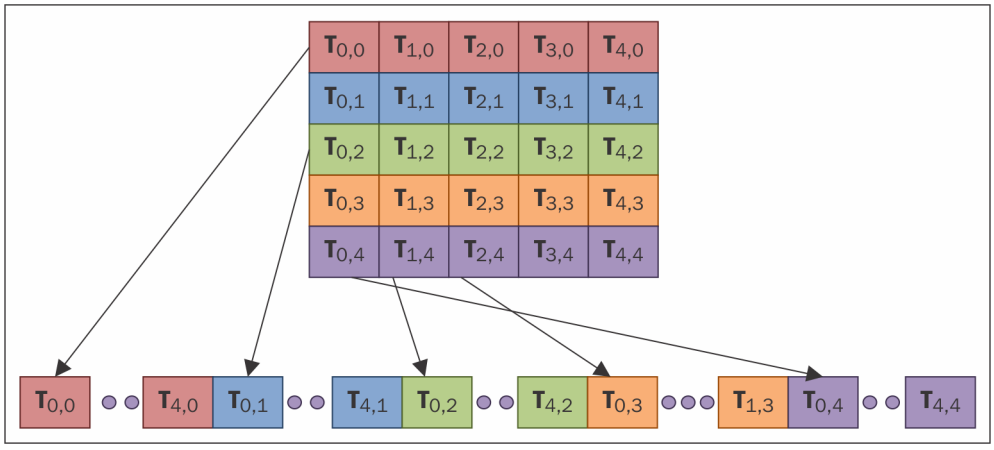
idx 参数是矩阵索引，用线程坐标 threadIdx.x 和 threadIdx.y 表示，如下图所示。然后，

索引为 idx 的矩阵元素乘以 2：

```
a[idx] *= 2;
```

注意，该内核函数将在 16 个不同的线程中分别执行一次。threadIdx.x 和 threadIdx.y 这两个变量包含从 0 到 3 的索引，每个线程中这对变量都不同。线程调度与 GPU 架构及其内部并发性直接相关。一个线程区块会被指派给一个流式多处理器（SM），然后这些线程被进一步划分为被称作 warp 的线程组，其大小由 GPU 的架构决定。同一线程组内的线程由一个叫作 warp 调度器的控制单元管理。为了充分利用 SM 本身的并发性，同一组内的线程必须执行相同的指令。如果不符合该条件，就出现了所谓的线程分歧（divergence of threads）。如果同一组内的线程执行不同的指令，控制单元将无法处理所有的 warp。它必须以线性模式对每个线程（或线程中同类型的子集）执行一系列指令。接下来我们看看线程区块如何被分成不同的 warp，线程又是如何根据 threadIdx 的值进行切分的。

threadIdx 结构体包含三个字段：threadIdx.x、threadIdx.y 和 threadIdx.z。



线程区块划分：T(x,y)，其中 x = threadIdx.x，y = threadIdx.y

我们看到，内核函数中的代码会被 CUDA 编译器 NVCC 自动编译。如果没有出错，将会创建指向被编译函数的指针。实际上，mod.get_function("doubleMatrix") 返回指向函数 func 的标识符：

```
func = mod.get_function("doubleMatrix")
```

要在设备上执行函数，必须先正确配置执行过程。这意味着需要确定用于辨别、区分属于不同区块的线程的坐标。可以使用 func 函数调用中的 block 参数实现：

```
func(a_gpu, block = (5, 5, 1))
```

在上述函数中，我们使用线性化的输入矩阵 a_gpu 和一个线程区块执行内核函数，线程区块在 *x* 方向有 5 个线程，*y* 方向有 5 个线程，*z* 方向有 1 个线程。这种结构划分在设计之初就考虑到了要并行实现算法。对工作量进行划分后会形成初步的并行形式，这也是利用 GPU 计算资源的充分必要条件。配置好内核调用参数之后，就可以调用内核函数，并在设备上并行执行指令了。每个线程执行相同的代码内核。

在 GPU 设备中计算完毕，我们使用一个数组来保存结果：

```
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
```

使用以下语句打印结果：

```
print a
print a_doubled
```

知识扩展

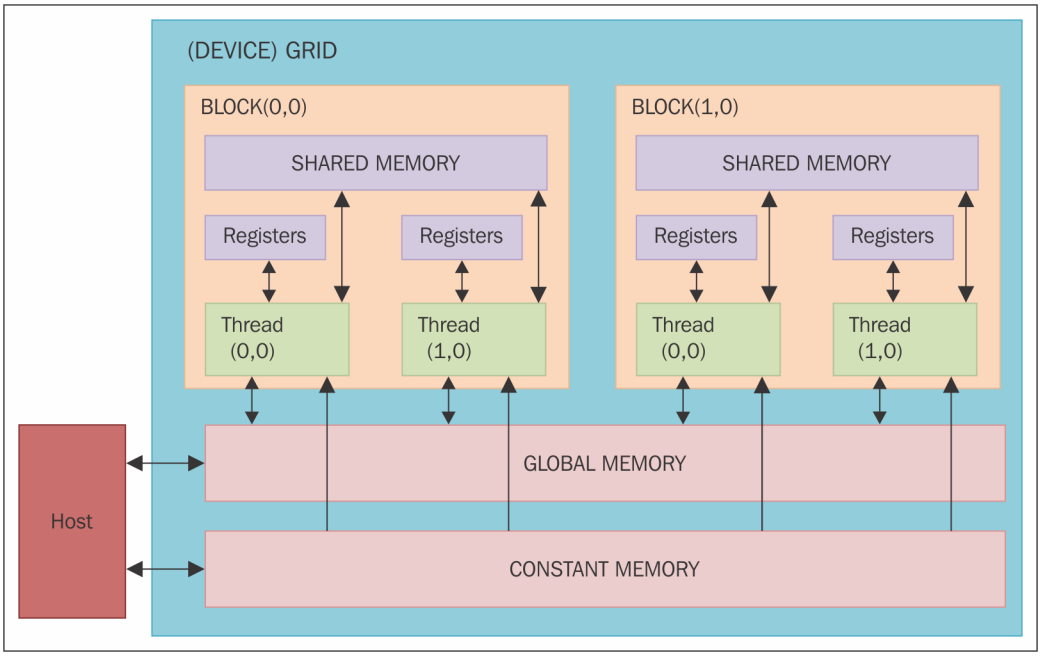
一个 warp 一次执行一个通用指令。因此，要最大化地提高这种结构的效率，必须使用同一个线程的执行流程。如果指定某个多处理器运行多个线程区块，会将线程区块分为多个 warp，并由 warp 调度器来实施调度。

通过矩阵操作理解 PyCUDA 内存模型

为了最大限度地利用可用资源，PyCUDA 程序应该遵守 SM 结构及其内部组织形式所要求的规则；SM 对于线程的性能会有限制。尤其是，了解并正确使用 GPU 提供的不同类型的内存，对于实现程序效率最大化至关重要。在支持 CUDA 的 GPU 显卡中，有 4 类内存，定义如下。

- **寄存器（registers）**：每个线程将被分配一个寄存器。每个线程只能访问自身的寄存器，而无法访问其他线程的寄存器，即使与对方同属于一个线程区块。
- **共享存储器（shared memory）**：在共享存储器中，每个线程区块都有一个其内部线程共享的内存。这部分内存速度极快。
- **常数存储器（constant memory）**：一个网格中的所有线程一直都可以访问这部分内存，但只能在读取时访问。常数存储器中的数据在应用持续期间一直存在。
- **全局存储器（global memory）**：所有网格中的线程（因此也包括所有内核）都可访问全局存储器，其中的常数存储器在应用持续期间一直存在。

GPU 的内存模型如下图所示。



GPU 内存模型

如果要让 PyCUDA 程序达到令人满意的性能，一个关键点就是要理解并非所有的内存都是一样的，必须充分利用每种类型的内存。基本的思路是通过使用共享存储器，尽量减少对全局存储器的访问。这种技巧通常用于区分问题的域 / 陪域（domain/codomain），好让线程区块在一个闭合的数据子集中进行运算。这样，属于该区块的线程将共同加载共享的全局存储器，用于在内存中进行运算，然后再充分利用这块内存区的高速特性。

每个线程要执行的基本步骤如下：

1. 从全局存储器中将数据加载至共享存储器。
2. 同步该区块中的全部线程，让每个线程可以读取共享存储器的安全位置，共享存储器中包含所有其他线程。
3. 处理共享存储器中的数据。
4. 如有必要，再次进行同步，确保共享存储器中已经获得最新的结果。
5. 在全局存储器中写入结果。

具体实现

为了更好地理解该技巧，我们来看一个例子。这个例子是求两个矩阵的乘积。下面的代码显示的是以标准方式进行的矩阵乘法，以及相应的线性代码，用于计算应该从矩阵数据的哪一

行哪一列加载元素：

```
void SequentialMatrixMultiplication(float*M,float *N,float *P, int width)
{
    for (int i=0; i< width; ++i)
        for(int j=0;j < width; ++j) {
            float sum = 0;
            for (int k = 0 ; k < width; ++k) {
                float a = M[I * width + k];
                float b = N[k * width + j];
                sum += a * b;
                        }
            P[I * width + j] = sum;
    }
}
```

如果让每个线程负责计算矩阵中的一个元素，那么内存访问将占用该算法的大部分执行时间。我们能做的是用一个线程区块来计算输出的一个子矩阵，这样就可以重复使用从全局存储器加载的数据，并实现各个线程之间的协作，最小化每个线程的内存访问。

以下示例说明了该技巧：

```
import numpy as np
from pycuda import driver, compiler, gpuarray, tools

# -- 初始化设备
import pycuda.autoinit

kernel_code_template =" ""
__global__ void MatrixMulKernel(float *a, float *b, float *c)
{
   int tx = threadIdx.x;
   int ty = threadIdx.y;
   float Pvalue = 0;
   for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
       float Aelement = a[ty * %(MATRIX_SIZE)s + k];
       float Belement = b[k * %(MATRIX_SIZE)s + tx];
       Pvalue += Aelement * Belement;
    }

   c[ty * %(MATRIX_SIZE)s + tx] = Pvalue;
}
" ""
MATRIX_SIZE = 5

a_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
```

```
b_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
c_cpu = np.dot(a_cpu, b_cpu)
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

c_gpu = gpuarray.empty((MATRIX_SIZE, MATRIX_SIZE), np.float32)

kernel_code = kernel_code_template % {
    'MATRIX_SIZE': MATRIX_SIZE
  }

mod = compiler.SourceModule(kernel_code)

matrixmul = mod.get_function("MatrixMulKernel")

matrixmul(
    a_gpu, b_gpu,
    c_gpu,
    block=(MATRIX_SIZE, MATRIX_SIZE, 1),
    )

# 打印结果
print "-" * 80
print "Matrix A (GPU):"
print a_gpu.get()

print "-" * 80
print "Matrix B (GPU):"
print b_gpu.get()

print "-" * 80
print "Matrix C (GPU):"
print c_gpu.get()

print "-" * 80
print "CPU-GPU difference:"
print c_cpu - c_gpu.get()

np.allclose(c_cpu, c_gpu.get())
```

以上示例的输出应该类似下面这样：

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\python
PyCudaMatrixManipulation.py
```

```
-------------------------------------------------------------------------
Matrix A (GPU):
[[ 0.90780383 -0.4782407   0.23222363 -0.63184392  1.05509627]
 [-1.27266967 -1.02834761 -0.15528528 -0.09468858  1.037099  ]
 [-0.18135822 -0.69884419  0.29881889 -1.15969539  1.21021318]
 [ 0.20939326 -0.27155793 -0.57454145  0.1466181   1.84723163]
 [ 1.33780348 -0.42343542 -0.50257754 -0.73388749 -1.883829  ]]
Matrix B (GPU):
[[ 0.04523897  0.99969769 -1.04473436  1.28909719  1.10332143]
 [-0.08900332 -1.3893919   0.06948703 -0.25977209 -0.49602833]
 [-0.6463753  -1.4424541  -0.81715286  0.67685211 -0.94934392]
 [ 0.4485206  -0.77086055 -0.16582981  0.08478995  1.26223004]
 [-0.79841441 -0.16199949 -0.35969591 -0.46809086  0.20455229]]
-------------------------------------------------------------------------
Matrix C (GPU):
[[-1.19226956  1.55315971 -1.44614291  0.90420711  0.43665022]
 [-0.73617989  0.28546685  1.02769876 -1.97204924 -0.65403283]
 [-1.62555301  1.05654192 -0.34626681 -0.51481217 -1.35338223]
 [-1.0040834   1.00310731 -0.4568972  -0.90064859  1.47408712]
 [ 1.59797418  3.52156591 -0.21708387  2.31396151  0.85150564]]
-------------------------------------------------------------------------
CPU-GPU difference:
[[  0.00000000e+00   0.00000000e+00   0.00000000e+00  -5.96046448e-08
    0.00000000e+00]
 [  0.00000000e+00   5.96046448e-08   0.00000000e+00   0.00000000e+00
    5.96046448e-08]
 [ -1.19209290e-07   2.38418579e-07   0.00000000e+00  -5.96046448e-08
    0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00  -2.98023224e-08  -5.96046448e-08
    0.00000000e+00]
 [  1.19209290e-07   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00]]
```

实例精解

我们来看一看 PyCUDA 编程工作流。首先，必须准备好输入矩阵以及用于保存结果的输出矩阵：

```
MATRIX_SIZE = 2
a_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
b_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
c_cpu = np.dot(a_cpu, b_cpu)
```

然后，使用 PyCUDA 函数 gpuarray.to_gpu() 将这些矩阵转移到 GPU 设备中：

```
a_gpu = gpuarray.to_gpu(a_cpu)
```

```
b_gpu = gpuarray.to_gpu(b_cpu)
c_gpu = gpuarray.empty((MATRIX_SIZE, MATRIX_SIZE), np.float32)
```

该算法的核心是下面这个内核函数：

```
__global__ void MatrixMulKernel(float *a, float *b, float *c)
{
   int tx = threadIdx.x;
   int ty = threadIdx.y;
   float Pvalue = 0;

   for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
        float Aelement = a[ty * %(MATRIX_SIZE)s + k];
        float Belement = b[k * %(MATRIX_SIZE)s + tx];
        Pvalue += Aelement * Belement;
    }

    c[ty * %(MATRIX_SIZE)s + tx] = Pvalue;
}
```

注意，__global__ 关键字表示该函数是一个内核函数，必须从主机上调用才能在设备上生成线程层级。

threadIdx.x 和 threadIdx.y 是网格中的线程索引。还需要注意，这些线程执行的是相同的内核代码，因此不同的线程会得到不同的线程坐标值。在这个并行代码中，线性版本代码（参看前面“具体实现”一节）中的循环变量 *i* 和 *j* 被替换为这两个线程索引 threadIdx.x 和 threadIdx.y。通过索引进行循环迭代，变成了通过这些线程索引进行，因此这里只有一次循环迭代。在调用 MatrixMulKernel 内核代码时，以 2×2 大小的并发线程网格形式执行：

```
mod = compiler.SourceModule(kernel_code)
matrixmul = mod.get_function("MatrixMulKernel")
matrixmul(
    a_gpu, b_gpu,
    c_gpu,
    block=(MATRIX_SIZE, MATRIX_SIZE, 1),
)
```

每次调用内核代码时，每个 CUDA 线程网格一般都由成千上百万个轻量级 GPU 线程组成。如果想创建足够的线程以充分利用硬件，通常要求做到大量的数据并行；例如，一个大数组的每个项可以通过另外一个线程进行计算。

最后，打印结果，验证计算结果是否正常，并报告 c_cpu 和 c_gpu 这两个矩阵乘积的差值：

```
print "-" * 80
print "CPU-GPU difference:"
print c_cpu - c_gpu.get()
```

```
np.allclose(c_cpu, c_gpu.get())
```

使用 GPUArray 调用内核

在上一个示例中，我们介绍了如何使用下面的类调用一个内核函数：

```
pycuda.compiler.SourceModule(kernel_source, nvcc="nvcc", options=None,
other_options)
```

这行代码从 CUDA 源代码 kernel_source 中创建一个模块。然后，使用给定的选项调用 NVIDIA 的 NVCC 编译器编译代码。

但是，PyCUDA 引入了 pycuda.gpuarray.GPUArray 类，提供了使用 CUDA 执行计算的高层接口：

```
class pycuda.gpuarray.GPUArray(shape, dtype, *, allocator=None, order="C")
```

其工作方式类似 numpy.ndarray，都是将数据保存至计算设备中并在该设备中进行计算。shape 和 dtype 参数与它们在 NumPy 中的使用方式一模一样。

GPUArray 类中的所有数值计算方法都支持广播标量数据类型。gpuarray 的创建很简单。一种方法是创建一个 NumPy 数组然后转换，如以下代码所示：

```
>>> import pycuda.gpuarray as gpuarray
>>> from numpy.random import randn
>>> from numpy import float32, int32, array
>>> x = randn(5).astype(float32)
>>> x_gpu = gpuarray.to_gpu(x)
```

可以通过正常方式打印 gpuarray：

```
>>> xarray([-0.24655211, 0.00344609, 1.45805557, 0.22002029,
1.28438667])
>>> x_gpuarray([-0.24655211,  0.00344609,  1.45805557,  0.22002029,
1.28438667])
```

具体实现

以下示例简单介绍了 GPUArray 的使用，也给出了 GPU 计算的一个常见使用场景，即作为其他计算的一个辅助步骤。脚本代码如下：

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
```

```
a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

输出（在 IDLE 中运行此函数）如下：

```
C:\Python Parallel Programming INDEX\Chapter 6 - GPU Programming wit
h Python\python PyCudaGPUArray.py
ORIGINAL MATRIX
[[-0.60254627  1.16694951  1.48510635 -1.46718287  2.11878467]
 [ 2.63159704 -3.6541729   2.44197178 -1.12101364  0.22178674]
 [-0.87713826 -1.9803952   0.98741448 -2.83859134 -1.55612338]
 [ 0.79552311 -0.25934356 -1.12207913 -0.21778747 -4.0459609 ]
 [-1.74858582  1.34928024 -2.55908132  2.22259712  0.82242775]]

DOUBLED MATRIX AFTER PyCUDA EXECUTION USING GPUARRAY CALL
[[-0.30127314  0.58347476  0.74255317 -0.73359144  1.05939233]
 [ 1.31579852 -1.82708645  1.22098589 -0.56050682  0.11089337]
 [-0.43856913 -0.9901976   0.49370724 -1.41929567 -0.77806169]
 [ 0.39776155 -0.12967178 -0.56103957 -0.10889374 -2.02298045]
 [-0.87429291  0.67464012 -1.27954066  1.11129856  0.41121387]]
```

实例精解

首先必须导入所有必需的模块：

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
```

输入矩阵 a_gpu 中包含随机生成的项。如果要在 GPU 中执行计算（将矩阵的所有项乘 2），只需要执行一行语句即可：

```
a_doubled = (2*a_gpu).get()
```

计算结果保存在 a_doubled 矩阵中（使用 get() 方法）。最后，打印结果：

```
print a_doubled
```

知识扩展

pycuda.gpuarray.GPUArray 类支持所有算术运算符，以及许多方法和函数，所有这些都是参照 NumPy 中的对应功能实现的。此外，pycuda.cumath 中还提供了许多特殊函数。可以

使用 pycuda.curandom 中的功能，生成近似于均匀分布随机数（uniformly distributed random numbers）组成的数组。

使用 PyCUDA 对逐元素表达式求值

PyCuda.elementwise.ElementwiseKernel 函数支持在复杂表达式上执行内核，这些复杂表达式由一个或多个操作数组成，共同构成一个计算步骤，具体如下：

```
ElementwiseKernel(arguments,operation,name,optional_parameters)
```

这里需要注意如下事项。

- arguments：这是一个 C 参数列表，包含内核执行涉及的所有参数。
- operation：这是要在给定参数上执行的操作。如果参数是向量，那么每一项都将执行所有操作。
- name：内核的名称。
- optional_parameters：这是以下示例中没有用到的编译指令。

具体实现

在本例中，我们将介绍 ElementwiseKernel 调用的常见用法。假设有由 50 个元素组成的两个向量 input_vector_a 和 input_vector_b，这两个向量都是随机构建的。我们的任务是要计算出它们的线性组合（linear combination）。

具体代码如下：

```
import pycuda.autoinit
import numpy
from pycuda.curandom import rand as curand
from pycuda.elementwise import ElementwiseKernel
import numpy.linalg as la


input_vector_a = curand((50,))
input_vector_b = curand((50,))
mult_coefficient_a = 2
mult_coefficient_b = 5


linear_combination = ElementwiseKernel(
    "float a, float *x, float b, float *y, float *c",
    "c[i] = a*x[i] + b*y[i]",
    "linear_combination")
```

```
linear_combination_result = gpuarray.empty_like(input_vector_a)
linear_combination(mult_coefficient_a, input_vector_a,
                   mult_coefficient_b, input_vector_b,
                   linear_combination_result)


print("INPUT VECTOR A =")
print(input_vector_a)

print("INPUT VECTOR B =")
print(input_vector_b)

print("RESULTING VECTOR C =")
print linear_combination_result

print("CHECKING THE RESULT EVALUATING THE DIFFERENCE VECTOR BETWEEN C AND
THE LINEAR COMBINATION OF A AND B")
print("C - (%sA + %sB) =" % (mult_coefficient_a, mult_coefficient_b))
print(linear_combination_result - (mult_coefficient_a*input_vector_a
                                   + mult_coefficient_b*input_
                                   vector_b))
assert la.norm((linear_combination_result -
                (mult_coefficient_a*input_vector_a +
                 mult_coefficient_b*input_vector_b)).get()) < 1e-5
```

上述代码在命令提示符中的输出如下：

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\ >python
PyCudaElementWise.py
INPUT VECTOR A =
[ 0.73191601  0.7004351   0.87159222  0.49621502  0.19640177
0.75579387
 0.35208538  0.97497243  0.36948711  0.34328628  0.06811771
0.04270195
 0.15690483  0.39899695  0.2927697   0.36201504  0.09503061
0.45646626
 0.35608584  0.01598917  0.75943208  0.49343511  0.79146844
0.33111155
 0.18454118  0.83971804  0.01466237  0.77959627  0.54659295
0.4575595
 0.55539894  0.23285247  0.14676388  0.72028935  0.87861985
0.13928016
 0.18071586  0.8029055   0.05551658  0.49400434  0.40941685
0.55373788
 0.07541087  0.55443048  0.19723719  0.72457349  0.46491891
0.65380263
```

```
 0.93845034  0.27472526]
INPUT VECTOR B =
[ 0.29464501  0.21645674  0.93407696  0.48678038  0.71135205
0.0588627
 0.99216938  0.879906    0.07517455  0.84360296  0.57358545
0.73907417
 0.06841258  0.1816148   0.53327322  0.30980903  0.96774238
0.90884209
 0.39139062  0.97678316  0.41284555  0.17893282  0.47421032
0.13706622
 0.62038481  0.22524452  0.67131585  0.06617502  0.02492006
0.99894243
 0.28288943  0.55505407  0.14323047  0.54854101  0.2742492
0.01146096
 0.45902726  0.03561942  0.78358203  0.32014725  0.13187674
0.42909116
 0.2633251   0.07679776  0.80823648  0.57373965  0.40740359
0.26024994
 0.61452144  0.46388686]
 RESULTING VECTOR C =
[ 2.93705702  2.48315382  6.41356945  3.42633176  3.94956398
1.80590129
  5.6650176   6.34947491  1.11484694  4.90458727  3.00416279
3.78077483
  0.65587258  1.70606792  3.25190544  2.2730751   5.02877283
5.45714283
  2.6691246   4.91589403  3.58309197  1.88153434  3.95398855
1.34755421
  3.47100639  2.80565882  3.38590407  1.89006758  1.21778619
5.90983152
  2.52524495  3.24097538  1.00968003  4.18328381  3.12848568
0.33586511
  2.65656805  1.78390813  4.02894306  2.58874488  1.47821736
3.25293159
  1.46744728  1.49284983  4.43565702  4.31784534  2.96685553
2.60885501
  4.94950771  2.86888456]
CHECKING THE RESULT EVALUATING THE DIFFERENCE VECTOR BETWEEN C AND THE
LINEAR COMBINATION OF A AND B
C - (2A + 5B) =
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.
 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.
 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

实例精解

在常规导入之后，我们注意到：

```
from pycuda.elementwise import ElementwiseKernel
```

必须先构建要操作的所有元素。记住，要完成的任务是求 input_vector_a 和 input_vector_b 这两个向量的线性组合。可以使用 PyCUDA 中的 curandom 库（用于生成伪随机数）初始化这两个向量。

使用以下代码导入该库：

```
from pycuda.curandom import rand as curand
```

然后定义随机向量（大小为 50 个元素）：

```
input_vector_a = curand((50,))
input_vector_b = curand((50,))
```

接下来定义两个乘法的系数，用于计算这两个向量的线性组合：

```
mult_coefficient_a = 2
mult_coefficient_b = 5
```

本例的核心是内核调用，为此我们使用 PyCUDA 的 ElementwiseKernel 构造函数，如下所示：

```
linear_combination = ElementwiseKernel(
    "float a, float *x, float b, float *y, float *c",
    "c[i] = a*x[i] + b*y[i]",
    "linear_combination")
```

第一行为参数列表（按 C 语言风格定义），定义了所有要加入到计算中的参数：

```
"float a, float *x, float b, float *y, float *c",
```

第二行定义了如何操作参数列表。对于索引为 i 的每个值，求以下组件之和：

```
"c[i] = a*x[i] + b*y[i]",
```

最后一行将 ElementwiseKernel 命名为 linear_combination。

在内核调用之后，定义最终保存结果的向量。它是一个和输入向量维度相同的空向量：

```
linear_combination_result = gpuarray.empty_like(input_vector_a)
# 最后对内核求值：
linear_combination(mult_coefficient_a, input_vector_a,
                   mult_coefficient_b, input_vector_b,
                   linear_combination_result)
```

可以使用以下代码检查结果：

```
assert la.norm((linear_combination_result -
                (mult_coefficient_a*input_vector_a +
                 mult_coefficient_b*input_vector_b)).get()) < 1e-5
```

assert 函数会检查结果，如果条件为假将引发错误。

知识扩展

除了源自 CUDA 库的 curand 库，PyCUDA 还提供了其他数学库，你可以在 http://documen.tician.de/pycuda 查看库列表。

使用 PyCUDA 进行 MapReduce 操作

PyCUDA 提供了在 GPU 上执行归约操作（reduction operation）的功能。可通过 pycuda.reduction.ReductionKernel 方法实现：

```
ReductionKernel(dtype_out, arguments, map_expr ,reduce_expr,
                name,optional_parameters)
```

这里要注意以下事项。

- dtype_out：这是输出的数据类型，必须通过 numpy.dtype 数据类型指定。
- arguments：这是一个 C 参数列表，包含了归约操作涉及的所有参数。
- map_expr：这是一个表示映射操作的字符串。该表达式中的每个向量必须通过变量 i 引用。
- reduce_expr：这是一个表示归约操作的字符串。这个表达式中的操作数以小写字母表示，如 a, b, c, ..., z。
- name：这是与 ReductionKernel 相关联的名称，内核使用该名称进行编译。
- optional_parameters：这些参数在该实例中并不重要，因为它们是编译器的指令。

上面的方法在向量参数上（至少一个）执行一个内核，在向量参数的每个项上执行 map_expr，然后在操作结果上执行 reduce_expr。

具体实现

本例介绍的是通过实例化 ReductionKernel 类求两个向量（500 个元素）的点积（dot product）。点积，也被称为标量积（scalar product），是一个代数运算，操作数是两个长度相等的数字序列（通常是坐标向量），结果为两个数字序列中对应项乘积之和。这是一个常见的 MapReduce 操作，map 操作是按索引求乘积，reduction 操作是求所有乘积之和。

完成这一任务的 PyCUDA 代码很短：

```
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
import numpy
from pycuda.reduction import ReductionKernel

vector_length = 400
input_vector_a = gpuarray.arange(vector_length, dtype=numpy.int)
input_vector_b = gpuarray.arange(vector_length, dtype=numpy.int)
dot_product = ReductionKernel(numpy.int,
                              arguments="int *x, int *y",
                              map_expr="x[i]*y[i]",
                              reduce_expr="a+b", neutral="0")

dot_product = dot_product(input_vector_a, input_vector_b).get()

print("INPUT VECTOR A")
print input_vector_a

print("INPUT VECTOR B")
print input_vector_b

print("RESULT DOT PRODUCT OF A * B")
print dot_product
```

在命令提示符中运行上述代码，将获得类似下面的输出：

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\>python
PyCudaReductionKernel.py


INPUT VECTOR A
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
```

```
 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251
 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269
 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287
 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305
 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323
 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341
 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377
 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395
 396 397 398 399]

INPUT VECTOR B
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251
 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269
 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287
 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305
 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323
 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341
 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377
 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395
 396 397 398 399]

RESULT DOT PRODUCT OF A * B
21253400
```

实例精解

在该脚本中，输入向量 input_vector_a 和 input_vector_b 都是整数向量。从上面的输出中可以看出，每个向量的元素的值在 0 到 399 之间（各有 400 个元素）：

```
vector_length = 400
```

```
input_vector_a = gpuarray.arange(vector_length, dtype=numpy.int)
input_vector_b = gpuarray.arange(vector_length, dtype=numpy.int)
```

在定义完输入之后，可以调用 PyCUDA 函数 ReductionKernel 来定义 MapReduce 操作：

```
dot_product = ReductionKernel(numpy.int,
                              arguments="int *x, int *y",
                              map_expr="x[i]*y[i]",
                              reduce_expr="a+b", neutral="0")
```

这个内核操作的定义如下：

- 参数列表中的第一项告诉我们，输出将是一个整数。
- 第二项以类似 C 语言的标记定义了输入（整数数组）的数据类型。
- 第三项是 map 操作，即求两个向量第 *i* 项元素的乘积。
- 第四个操作是 reduction 操作，即求所有乘积之和。

注意，调用 ReductionKernel 实例的最终结果是一个仍存在于 GPU 中的 GPUArray 标量。可以调用其 get 方法将其放到 CPU 中，或者也可以在 GPU 中使用这个值。

然后，按如下方式调用内核函数：

```
dot_product = dot_product (input_vector_a, input_vector_b).get()
```

最后，打印出输入向量以及最终的点积：

```
print input_vector_a
print input_vector_b
print dot_product
```

使用 NumbaPro 进行 GPU 编程

NumbaPro 是一个 Python 编译器，提供了基于 CUDA 的 API 编程接口，可以编写 CUDA 程序。它专门设计用来执行与数组相关的计算任务，和广泛使用的 NumPy 库类似。与数组相关的计算任务中的数据并行性非常适合由 GPU 等加速器完成。NumbaPro 可接受 NumPy 数组类型，并用这些数组生成可在 GPU 或多核 CPU 上执行的高效编译代码。

该编译器支持对 Python 函数指定类型签名（type signature），可开启运行时编译特性（被称为 JIT 编译）。

其中最重要的装饰器如下所示。

- numbapro.jit：支持开发者编写类 CUDA 的函数。遇到该装饰器时，编译器会将装饰器下的代码编译为伪汇编语言 PTX，在 GPU 中执行。

- numbapro.autojit：注释一个函数为延迟编译（deferred compilation），这意味着带有该签名的每个函数只会被编译一次。
- numbapro.vectorize：创建一个所谓的 ufunc 对象（NumPy 中的通用函数），接受一个函数，并以向量参数并行执行。
- guvectorize：创建所谓的 gufunc 对象（NumPy 中的泛通用函数）。这种对象可以操作整个子数组（参看 http://docs.continuum.io/numbapro/generalizedufuncs.html 获取更多资料）。

这些装饰器都有一个叫目标（target）的编译器指令，用于选择代码生成目标。NumbaPro 编译器支持并行和 GPU 等目标。并行目标可用于向量化操作，而 GPU 指令则将计算负载交给 NVIDIA 的 CUDA GPU 进行。

准备工作

NumbaPro 是 Anaconda Accelerate 的一部分，后者是 Continuum Analytics 公司的商业许可产品（学术用户可以免费协议使用 NumbaPro）。它基于以 BSD 协议发布的开源项目 Numba，而 Numba 自身也很大程度依赖于 LLVM 编译器。NumbaPro 的 GPU 后端利用了基于 LLVM 的 NVIDIA 编译器 SDK。

如想使用 NumbaPro，第一步要下载并安装 Anaconda 提供的 Python 发行版（http://continuum.io/downloads），这是一个完全免费的企业级 Python 发行版，可用于大规模数据处理、预测分析和科学计算，包含许多流行库（NumPy、SciPy、IPython 等），还提供了一个强大的包管理器 conda。

安装好 Anaconda 之后，从 Anaconda 的命令提示符中输入以下指令：

```
> conda update conda
> conda install accelerate
> conda install numbapro
```

NumbaPro 并没有包含 CUDA 驱动器，应由用户确保系统已经使用了最新的驱动器。在安装完后，可以检测 CUDA 库和 GPU。我们从 Anaconda 控制台打开 Python 并输入：

```
import numbapro
numbapro.check_cuda()
```

这两行代码的输出如下所示（我们使用的是 64 位的 Anaconda 发行版）：

```
C:\Users\Giancarlo\Anaconda>python
Python 2.7.10 |Anaconda 2.3.0 (64-bit)| (default, May 28 2015, 16:44:52)
[MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
```

```
Please check out: http://continuum.io/thanks and https://binstar.org
>>> import numbapro
Vendor:  Continuum Analytics, Inc.
Package: mkl
Message: trial mode expires in 30 days
Vendor:  Continuum Analytics, Inc.
Package: mkl
Message: trial mode expires in 30 days
Vendor:  Continuum Analytics, Inc.
Package: numbapro
Message: trial mode expires in 30 days
>>> numbapro.check_cuda()
-----------------------------libraries detection-----------------------
Finding cublas
        located at C:\Users\Giancarlo\Anaconda\DLLs\cublas64_60.dll
        trying to open library...       ok
Finding cusparse
        located at C:\Users\Giancarlo\Anaconda\DLLs\cusparse64_60.dll
        trying to open library...       ok
Finding cufft
        located at C:\Users\Giancarlo\Anaconda\DLLs\cufft64_60.dll
        trying to open library...       ok
Finding curand
        located at C:\Users\Giancarlo\Anaconda\DLLs\curand64_60.dll
        trying to open library...       ok
Finding nvvm
        located at C:\Users\Giancarlo\Anaconda\DLLs\nvvm64_20_0.dll
        trying to open library...       ok
        finding libdevice for compute_20...     ok
        finding libdevice for compute_30...     ok
        finding libdevice for compute_35...     ok
-----------------------------hardware detection-----------------------
Found 1 CUDA devices
id 0            GeForce 840M                                [SUPPORTED]
                       compute capability: 5.0
                           pci device id: 0
                              pci bus id: 8
Summary:
        1/1 devices are supported
PASSED
True
>>>
```

具体实现

在本例中，我们使用注释 @guvectorize 来说明 NumbaPro 编译器的用法。在下面的任务中，我们试着使用 NumbaPro 模块执行矩阵乘法：

```
from numbapro import guvectorize
import numpy as np


@guvectorize(['void(int64[:,:], int64[:,:], int64[:,:])'],
             '(m,n),(n,p)->(m,p)')
def matmul(A, B, C):
    m, n = A.shape
    n, p = B.shape
    for i in range(m):
        for j in range(p):
            C[i, j] = 0
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]


dim = 10
A = np.random.randint(dim, size=(dim, dim))
B = np.random.randint(dim, size=(dim, dim))


C = matmul(A, B)
print("INPUT MATRIX A")
print(":\n%s" % A)
print("INPUT MATRIX B")
print(":\n%s" % B)
print("RESULT MATRIX C = A*B")
print(":\n%s" % C)
```

运行上述代码（使用 Anaconda 控制台）之后，应该会出现类似如下的输出：

```
INPUT MATRIX A
:
[[7 7 8 5 8 5 1 9 5 9]
 [3 5 5 4 6 7 6 5 3 1]
 [7 1 6 8 7 9 0 3 3 3]
 [7 4 4 3 7 8 1 2 1 2]
 [4 7 7 1 3 5 5 6 7 6]
 [5 0 1 5 8 4 4 4 4 9]
 [1 3 2 0 7 3 7 2 3 4]
 [0 2 9 0 7 5 9 7 4 7]
```

```
 [7 3 7 6 5 6 4 2 2 7]
 [2 1 9 7 1 0 3 5 7 3]]
INPUT MATRIX B
:
[[2 9 8 4 2 3 9 7 3 1]
 [9 1 3 3 8 0 7 6 3 5]
 [7 4 9 6 6 5 9 7 6 6]
 [6 8 3 1 5 4 4 7 7 5]
 [6 2 5 1 2 8 6 0 5 8]
 [4 4 5 7 6 0 1 1 3 8]
 [2 7 8 6 1 9 8 4 1 6]
 [2 2 9 8 3 6 1 4 7 4]
 [9 9 6 9 3 3 3 2 4 9]
 [8 4 6 7 8 8 8 6 7 8]]

RESULT MATRIX C = A*B
:
[[368 284 402 331 304 295 361 291 327 378]
 [231 207 278 226 188 199 236 177 193 273]
 [248 247 280 217 208 190 243 198 232 279]
 [201 181 232 175 173 149 218 156 170 225]
 [297 239 331 301 239 225 290 225 229 315]
 [235 229 270 222 181 248 246 175 219 280]
 [174 142 201 166 124 185 192 108 129 217]
 [267 213 348 297 212 292 289 194 233 334]
 [266 254 305 239 228 230 303 234 232 288]
 [227 219 255 215 166 189 214 196 204 229]]
```

实例精解

@guvectorize 注释适用于数组参数。该装饰器可接受一个额外参数，指定 gufunc 签名。各个参数的解释如下所示。

- 前三个参数指定要管理的数据类型，为整数数组：'void(int64[:,:]、int64[:,:]、int64[:,:])'。
- @guvectorize 的最后一个参数指定如何操作矩阵维度：'(m,n),(n,p)->(m,p)'。

```
@guvectorize(['void(int64[:,:], int64[:,:], int64[:,:])'],
             '(m,n),(n,p)->(m,p)')
```

在之后的代码中，我们定义 matmul(A, B, C) 操作。它接受两个输入矩阵 ***A*** 和 ***B***，生成输出矩阵 ***C***。根据 gufunc 签名，可以知道：

A(m,n)* B(n,p) = C(m,p)，其中 m、n、p 为矩阵的维度

矩阵乘积只需执行三个带矩阵索引的 for 循环即可求出：

```
for i in range(m):
    for j in range(p):
        C[i, j] = 0
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
```

NumPy 中的 randint 函数可用于从随机矩阵构建整数：

```
dim = 10
A = np.random.randint(dim, size=(dim, dim))
B = np.random.randint(dim, size=(dim, dim))
```

最后，使用这些矩阵作为参数调用 matmul 函数，并打印最终的矩阵：

```
C = matmul(A, B)
print("INPUT MATRIX A")
print("RESULT MATRIX C = A*B")
     print(":\n%s" % C)
```

通过 NumbaPro 使用 GPU 加速的库

NumbaPro 提供了一个 CUDA 库的 Python 封装，可用于数值计算（numerical computing）。使用了这些库后，无须编写任何与 GPU 相关的代码即可获得大幅速度提升。相关库的解释如下所示。

- **cuBLAS**：这是 NVIDIA 开发的一个库，提供了可在 GPU 上运行的主要线性代数函数。与在 CPU 上实现了线性代数函数的基础线性代数子程序库（Basic Linear Algebra Subprograms，简称 BLAS）类似，cuBLAS 库将其函数划分为以下三个层次。
  - **第一层**：向量操作。
  - **第二层**：矩阵和向量之间的事务处理（transaction）。
  - **第三层**：矩阵之间的操作。

  将函数划分为这三层，是基于执行选定操作需要多少层嵌套循环决定的。更准确地说，每个层次的操作是执行完选定函数所需要的基本循环。
- **cuFFT**：该库提供了一个在 NVIDIA GPU 上以分布式方式计算快速傅里叶变换（Fast Fourier Transform，FFT）的简单接口，可以在不自行实现 FFT 的情况下利用 GPU 的并行特性。
- **cuRAND**：该库支持创建准随机数。准随机数指的是由一个确定性算法生成的随机数。
- **cuSPArse**：该库提供了一组用于管理稀疏矩阵（sparse matrix）的函数。和第一个库不同，它的函数划分为四个层次。

- 第一层：这些是稀疏向量（sparse vector）和稠密向量（dense vector）之间的操作。
- 第二层：这些是稀疏矩阵（sparse matrix）和稠密向量（dense vector）之间的操作。
- 第三层：这些是稀疏矩阵（sparse matrix）和一组稠密向量（dense vector）之间的操作。
- 转换：这些是支持不同存储格式之间转换的操作。

具体实现

在本例中，我们介绍通用矩阵乘（General Matrix Multiply，简称 GEMM）的一种实现，通用矩阵乘是一种在 NVIDIA GPU 上执行矩阵间乘法（matrix-matrix multiplication）的例行程序。下面将分别解释使用 NumPy 模块的线性版与使用 cuBLAS 库的并行版。另外，还将对比两种算法之间的执行时间。

本例的代码如下所示：

```
import numbapro.cudalib.cublas as cublas
import numpy as np
from timeit import default_timer as timer


dim = 10


def gemm():
    print("Version 2".center(80, '='))

    A = np.random.rand(dim, dim)
    B = np.random.rand(dim, dim)
    D = np.zeros_like(A, order='F')

    print("MATRIX A :")
    print A
    print("VECTOR B :")
    print B

    # NumPy
    start = timer()
    E = np.dot(A, B)
    numpy_time = timer() - start
    print("Numpy took %f seconds" % numpy_time)

    # cuBLAS
    blas = cublas.Blas()

    start = timer()
```

```
    blas.gemm('T', 'T', dim, dim, dim, 1.0, A, B, 1.0, D)
    cuda_time = timer() - start
    print ("RESULT MATRIX EVALUATED WITH CUBLAS")
    print D
    print("CUBLAS took %f seconds" % cuda_time)
    diff = np.abs(D - E)
    print("Maximum error %f" % np.max(diff))


def main():

    gemm()

if __name__ == '__main__':
    main()
```

运行此例获得的输出如下：

```
MATRIX A :
[[ 0.79582178  0.95671563  0.69251157  0.85600979  0.32826726  0.72861569
   0.20724061  0.55065641  0.2257875   0.90146437]
 [ 0.6742022   0.43449657  0.04862685  0.9023226   0.87598306  0.20774405
   0.15774015  0.2847742   0.81601615  0.34114773]
 [ 0.61500219  0.65982283  0.73493152  0.21913261  0.80862566  0.73982082
   0.84005388  0.38745489  0.676947    0.31530397]
 [ 0.60694411  0.65138528  0.63773284  0.06589098  0.49177294  0.02029247
   0.9064746   0.93419845  0.14609622  0.28317855]
 [ 0.60166404  0.41423776  0.09938464  0.19315303  0.07374789  0.45335697
   0.2912572   0.81481984  0.65222424  0.0670377 ]
 [ 0.32192297  0.30244072  0.86595209  0.37701833  0.79095644  0.11518194
   0.88491826  0.98290063  0.62965353  0.38323725]
 [ 0.21512101  0.64731098  0.4079146   0.8371392   0.01398673  0.85945652
   0.0586854   0.48812094  0.3625991   0.58142603]
 [ 0.77378663  0.43994483  0.5620805   0.70350504  0.60589009  0.09605428
   0.25423268  0.06869655  0.13642323  0.00221422]
 [ 0.77808301  0.47386303  0.54323866  0.42010733  0.80652762  0.05903843
   0.63316824  0.58479485  0.45141828  0.46231481]
 [ 0.97122802  0.53723365  0.68688748  0.54315409  0.00883411  0.9855186
   0.53542786  0.83478941  0.27459888  0.21024639]]
VECTOR B :
[[ 0.17084153  0.44546677  0.21551063  0.39731923  0.00102686  0.81069924
   0.00681474  0.01126972  0.13769525  0.63437229]
 [ 0.81913609  0.97583768  0.52579565  0.20179695  0.24066758  0.18154282
   0.75033104  0.41878918  0.96892428  0.54358419]
 [ 0.10071768  0.3090773   0.94185921  0.70550442  0.10651627  0.62659408
```

```
    0.23255164  0.96166165  0.65615938  0.16991118]
 [ 0.84163163  0.59296382  0.12281989  0.32851275  0.78716318  0.02568872
    0.02367708  0.65485736  0.79834789  0.76747705]
 [ 0.90406949  0.03424157  0.01519989  0.5011444   0.63175281  0.17705116
    0.16257016  0.81357471  0.58567631  0.24503327]
 [ 0.62989968  0.47944669  0.86860435  0.94086568  0.24312278  0.13450463
    0.16352136  0.42323191  0.46907905  0.97772097]
 [ 0.44608094  0.19969488  0.01035155  0.69528549  0.07219375  0.91454669
    0.18330497  0.76095336  0.12880003  0.24301603]
 [ 0.37860881  0.33079438  0.19275564  0.58316669  0.35753971  0.63697732
    0.72063491  0.42698316  0.53811423  0.83682958]
 [ 0.42135462  0.89413827  0.00620849  0.63770542  0.29376823  0.68415057
    0.71826696  0.9748898   0.9086774   0.7084634 ]
 [ 0.08020851  0.47789158  0.45538401  0.26468263  0.84960276  0.1108932
    0.0407631   0.41811299  0.2539022   0.73346706]]
Numpy took 1.167435 seconds
RESULT MATRIX EVALUATED WITH CUBLAS
[[ 2.93393517  3.22653293  2.58999843  2.97688025  2.40723642  2.22561846
    1.71083261  3.20145366  3.4654546   3.9246803 ]
 [ 2.70759988  2.42236864  0.94108333  2.20715685  2.06739391  1.78390442
    1.37381915  2.80760808  2.87826551  2.88739456]
 [ 2.93301949  2.70921232  2.08465713  3.39447429  1.76684939  2.84034554
    1.8600905   3.70096673  3.21368161  3.20257798]
 [ 2.05665894  1.92477247  1.42646422  2.45288009  1.27576149  2.65682509
    1.68187918  2.6942483   2.30742661  2.35163885]
 [ 1.68553937  1.98030198  1.05436088  2.03107385  0.98066787  1.94328559
    1.54050405  1.8876191   2.04514196  2.49719893]
 [ 2.55782414  2.2600454   1.57942935  3.11991574  1.91570669  2.93236718
    1.92525406  3.76932667  3.03618471  2.87628333]
 [ 2.27705425  2.53777179  1.98218876  2.30511984  1.85547257  1.36423334
    1.39131705  2.43879465  2.75148098  3.14994564]
 [ 1.94662205  1.62822264  1.12425671  1.72230283  1.21131853  1.56748417
    0.79113948  2.08449619  2.05742732  1.82536594]
 [ 2.42686338  2.22641127  1.3762425   2.57727754  1.80747335  2.53040609
    1.51847658  3.05078902  2.68199133  2.72340269]
 [ 2.44854528  2.69315101  2.3255071   3.17886105  1.47260987  2.69597578
    1.65043895  2.79595207  2.82714486  3.58489296]]
CUBLAS took 0.004226 seconds
Maximum error 0.000000
```

运行结果证实 cuBLAS 库的效率更高。

实例精解

为了比较矩阵乘积的 NumPy 和 cuBLAS 实现，我们导入所有必需的库：

```
import numbapro.cudalib.cublas as cublas
import numpy as np
```

同时，定义矩阵维度：

```
dim = 10
```

核心算法是 gemm() 函数。首先，定义输入矩阵：

```
A = np.random.rand(dim,dim)
B = np.random.rand(dim,dim)
```

这里，用 D 保存 cuBLAS 实现的输出：

```
D = np.zeros_like(A, order='F')
```

在本例中，我们比较通过 NumPy 和 cuBLAS 完成的计算。NumPy 求值表达式为：E = np.dot(A,B)，其中矩阵 E 将保存点积。

最后，cuBLAS 实现如下：

```
blas = cublas.Blas()
    start = timer()
    blas.gemm('T', 'T', dim, dim, dim, 1.0, A, B, 1.0, D)
    cuda_time = timer() - start
```

gemm() 函数是 cuBLAS 中的一个第三层函数：

```
numbapro.cudalib.cublas.Blas.gemm(transa, transb, m, n, k, alpha,
                                  A, B,beta, C)
```

它以如下形式实现了矩阵间乘法：

```
C = alpha * op(A) * op(B) + beta * C，其中 op 为转置操作（transpose）或非运算
```

在函数的最后，比较两个结果，并报告执行时间（cuda_time）：

```
print("CUBLAS took %f seconds" % cuda_time)
    diff = np.abs(D - E)
    print("Maximum error %f" % np.max(diff))
```

知识扩展

在此例中，我们学习了 cuBLAS 库的应用。更完整的资料，请查看 http://docs.nvidia.com/cuda/cublas/index.html 和 http://docs.continuum.io/numbapro/cudalib，以获取 NumbaPro 包装的完整 CUDA 函数库列表。

使用 PyOpenCL 模块

开放计算语言（Open Computing Language，简称 OpenCL）是用于开发跨不同平台运行程序的框架，平台可以是使用不同生产商生产的 CPU 或 GPU 组成的。该框架由 Apple 创建，后来由一个称为 Khronos Group 的非营利联合体开发和维护。这个框架是在 GPU 上使用 CUDA 执行软件的主要替代方案，但其出发点截然不同。CUDA 的优点在于专精（由 NVIDIA 生产、开发并与其产品兼容），以可移植性不高为代价确保了极佳的性能。而 OpenCL 则给出了与市场上几乎所有设备兼容的解决方案。用 OpenCL 编写的软件可以在所有主流厂商（如 Intel、NVIDIA、IBM 和 AMD）生产的处理器上运行。OpenCL 包含一种基于 C99（有部分限制）的语言，可编写内核，支持直接和 CUDA-C-Fortran 或 CUDA 相同的方式使用硬件。OpenCL 提供了运行高度并行、同步的原语，如内存区域指示器和不同执行平台的控制机制。但是 OpenCL 程序的可移植性也有限制，只能在不同设备上运行相同的代码，这也确保了各个平台上的性能同等可靠。为了获得最佳性能，根据设备的特征对代码进行优化至关重要。在以下示例中，我们将探讨 OpenCL 在 Python 中的实现，叫作 PyOpenCL 。

准备工作

PyOpenCL 之于 OpenCL，好比 PyCUDA 之于 CUDA，是 GPGPU 平台的 Python 包装器（PyOpenCL 在 NVIDIA 和 AMD 两家的 GPU 显卡下均可运行），由 Andreas Klöckner 开发和维护。如使用 Christoph Gohlke 提供的二进制包，在 Windows 平台下安装 PyOpenCL 很容易。它的网页上含有数百种 Python 软件包最新版本的 Windows 二进制安装包，对于使用 Windows 系统的 Python 用户来说非常有帮助。

按照如下说明，可以在具备 NVIDIA GPU 显卡的 Windows 电脑中，为 Python 2.7 发行版构建 32 位的 PyOpenCL 库 ：

1. 前往 `http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyopencl`，并单击 `pyopencl-2015.1-cp27-none-win32.whl` 下载文件（以及相关依赖包，如果需要的话）。
2. 从 `http://registrationcenter.intel.com/irc_nas/5198/opencl_runtime_15.1_x86_setup.msi` 下载并安装 Win32 的 OpenCL 驱动器（Intel 提供）。
3. 最后，从命令提示符输入以下命令，安装 `pyOpenCL` ：

```
pip install pyopencl-2015.1-cp27-none-win32.whl
```

具体实现

在第一个示例中，我们验证是否正确安装 PyOpenCL 环境。

因此，下面给出一个简单脚本，枚举使用 OpenCL 库可获得的主要硬件特征 ：

```
import pyopencl as cl


def print_device_info():
    print('\n' + '=' * 60 + '\nOpenCL Platforms and Devices')
    for platform in cl.get_platforms():
        print('=' * 60)
        print('Platform - Name:  ' + platform.name)
        print('Platform - Vendor:  ' + platform.vendor)
        print('Platform - Version:  ' + platform.version)
        print('Platform - Profile:  ' + platform.profile)
        for device in platform.get_devices():
            print('    ' + '-' * 56)
            print('    Device - Name:  '
                  + device.name)
            print('    Device - Type:  '
                  + cl.device_type.to_string(device.type))
            print('    Device - Max Clock Speed:  {0} Mhz'
                  .format(device.max_clock_frequency))
            print('    Device - Compute Units:  {0}'
                  .format(device.max_compute_units))
            print('    Device - Local Memory:  {0:.0f} KB'
                  .format(device.local_mem_size/1024.0))
            print('    Device - Constant Memory:  {0:.0f} KB'
                  .format(device.max_constant_buffer_size/1024.0))
            print('    Device - Global Memory: {0:.0f} GB'
                  .format(device.global_mem_size/1073741824.0))
            print('    Device - Max Buffer/Image Size: {0:.0f} MB'
                  .format(device.max_mem_alloc_size/1048576.0))
            print('    Device - Max Work Group Size: {0:.0f}'
                  .format(device.max_work_group_size))

    print('\n')

if __name__ == "__main__":
    print_device_info()
```

此脚本的输出中会显示已安装 CPU 和 GPU 显卡的主要特征，类似如下所示：

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\>python
PyOpenCLDeviceInfo.py


============================================================
OpenCL Platforms and Devices
============================================================
```

```
Platform - Name:  NVIDIA CUDA
Platform - Vendor:  NVIDIA Corporation
Platform - Version:  OpenCL 1.1 CUDA 6.0.1
Platform - Profile:  FULL_PROFILE
    ---------------------------------------------------
    Device - Name:  GeForce GT 240
    Device - Type:  GPU
    Device - Max Clock Speed:  1340 Mhz
    Device - Compute Units:  12
    Device - Local Memory:  16 KB
    Device - Constant Memory:  64 KB
    Device - Global Memory: 1 GB




============================================================
Platform - Name:  Intel(R) OpenCL
Platform - Vendor:  Intel(R) Corporation
Platform - Version:  OpenCL 1.2
Platform - Profile:  FULL_PROFILE
  -------------------------------------------------------
  Device - Name:  Intel(R) Core(TM)2 Duo CPU
  Device - Type:  CPU
  Device - Max Clock Speed:  2330 Mhz
  Device - Compute Units:  2
  Device - Local Memory:  32 KB
  Device - Constant Memory:  128 KB
  Device - Global Memory: 2 GB
```

实例精解

上面的代码很简单。在第一行中，我们导入 pyopencl 模块：

```
import pyopencl as cl
```

然后，使用 platform.get_devices() 方法获取设备列表。接着，在屏幕上打印每个设备的主要特征：

- 名称和设备类型
- 最大转速（clock speed）
- 计算单元（compute unit）
- 本地 / 常数 / 全局存储器

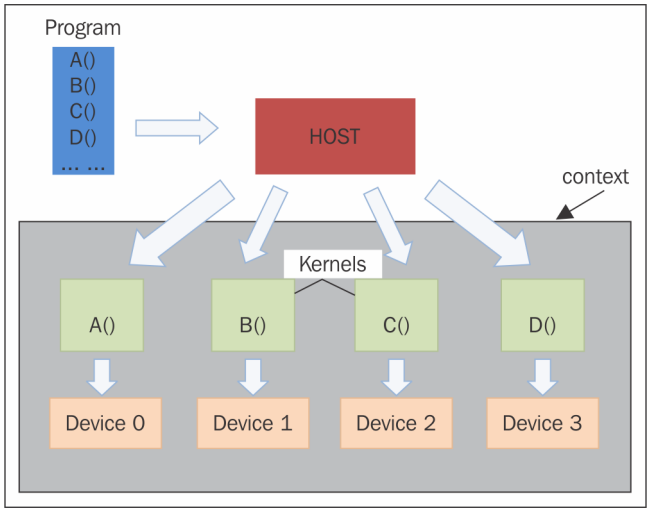
如何构建一个 PyOpenCL 应用

与使用 PyCUDA 进行编程一样，构建 PyOpenCL 应用的第一步是编码宿主应用（host application）。实际上，这个应用将在宿主电脑（通常为用户的电脑）上执行，然后调度连接设备（GPU 显卡）上的内核应用。

宿主应用必须包含 5 个数据结构。

- **设备**：指出用来执行内核代码的硬件。PyOpenCL 应用可以在 CPU 和 GPU 显卡上执行，也可以嵌入到设备中，如现场可编程门阵列（Field Programmable Gate Array，简称 FPGA）。
- **程序**：这指的是一组内核。程序将选择必须在设备上执行的内核。
- **内核**：这是将在设备上执行的代码。内核本质上是一个类 C 函数，可以在支持 OpenCL 驱动器的任意设备上编译执行。内核是宿主调用在设备上运行的函数的唯一方法。宿主调用内核时，设备上的许多工作项将开始运行。每个工作项都将运行内核代码，但是操作的是数据集的不同部分。
- **命令队列（command queue）**：每个设备通过该数据结构接收内核。命令队列用于确定设备上内核的执行顺序。
- **上下文（context）**：这指的是一组设备。上下文支持设备接收内核，转移数据。

下图说明了这些数据结构如何在宿主应用中运作。注意，一个程序可以包含多个在设备上执行的函数，每个内核都只封装了程序中的一个函数。



PyOpenCL 编程

具体实现

在此例中，我们将介绍构建一个 PyOpenCL 程序的基本步骤。此示例中的任务是对两个向量并行求和（parallel sum）。为了确保输出可读，假设有两个包含 100 个元素的向量。输出向量中的第 *i* 个元素，将是 vector_a 和 vector_b 向量中第 *i* 个元素之和。

当然，为了明显地体现并行执行的效果，可以将 vector_dimension 输入参数提升几个数量级：

```
import numpy as np
import pyopencl as cl
import numpy.linalg as la

vector_dimension = 100

vector_a = np.random.randint(vector_dimension, size=vector_dimension)
vector_b = np.random.randint(vector_dimension, size=vector_dimension)

platform = cl.get_platforms()[0]
device = platform.get_devices()[0]

context = cl.Context([device])
queue = cl.CommandQueue(context)

mf = cl.mem_flags
a_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=vector_a)
b_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=vector_b)

program = cl.Program(context," ""
__kernel void vectorSum(__global const int *a_g, __global const int
*b_g, __global int *res_g) {
 int gid = get_global_id(0);
 res_g[gid] = a_g[gid] + b_g[gid];
}
" "").build()

res_g = cl.Buffer(context, mf.WRITE_ONLY, vector_a.nbytes)
program.vectorSum(queue, vector_a.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(vector_a)
cl.enqueue_copy(queue, res_np, res_g)

print ("PyOPENCL SUM OF TWO VECTORS")
print ("Platform Selected = %s" %platform.name )
```

```
print ("Device Selected = %s" %device.name)
print ("VECTOR LENGTH = %s" %vector_dimension)
print ("INPUT VECTOR A")
print vector_a
print ("INPUT VECTOR B")
print vector_b
print ("OUTPUT VECTOR RESULT A + B")
print res_np

assert(la.norm(res_np - (vector_a + vector_b))) < 1e-5
```

命令提示符中的输出应该类似下面这样：

```
C:\Python CookBook\ Chapter 6 - GPU Programming with Python\Chapter 6 -
codes>python PyOpenCLParallellSum.py

Platform Selected = NVIDIA CUDA
Device Selected = GeForce GT 240

VECTOR LENGTH = 100
INPUT VECTOR A
[ 0 29 88 46 68 93 81  3 58 44 95 20 81 69 85 25 89 39 47 29 47 48 20 86
59 99  3 26 68 62 16 13 63 28 77 57 59 45 52 89 16  6 18 95 30 66 19 29
31 18 42 34 70 21 28  0 42 96 23 86 64 88 20 26 96 45 28 53 75 53 39 83
85 99 49 93 23 39  1 89 39 87 62 29 51 66  5 66 48 53 66  8 51  3 29 96
67 38 22 88]

INPUT VECTOR B
[98 43 16 28 63  1 83 18  6 58 47 86 59 29 60 68 19 51 37 46 99 27  4 94
5 22 3 96 18 84 29 34 27 31 37 94 13 89  3 90 57 85 66 63  8 74 21 18 34
93 17 26  9 88 38 28 14 68 88 90 18  6 40 30 70 93 75  0 45 86 15 10 29
 84 47 74 22 72 69 33 81 31 45 62 81 66 69 14 71 96 91 51 35 4 63 36 28
65 10 41]

OUTPUT VECTOR RESULT A + B
[ 98  72 104  74 131  94 164  21  64 102 142 106 140  98 145  93 108  90
  84  75 146  75  24 180  64 121   6 122  86 146  45  47  90  59 114 151
  72 134  55 179  73  91  84 158  38 140  40  47  65 111  59  60  79 109
  66  28  56 164 111 176  82  94  60  56 166 138 103  53 120 139  54  93
 114 183  96 167  45 111  70 122 120 118 107  91 132 132  74  80 119 149
 157  59  86   7  92 132  95 103  32 129]
```

实例精解

导入所需的模块之后，我们定义输入向量：

```
vector_dimension = 100
vector_a = np.random.randint(vector_dimension, size= vector_dimension)
vector_b = np.random.randint(vector_dimension, size= vector_dimension)
```

每个向量包含通过 NumPy 函数 np.random.randint(max integer,size of the vector) 随机选择的 100 个整数项。

然后，选择运行内核代码的设备。为此，必须首先使用 PyOpenCL 的 get_platform() 语句选择平台：

```
platform = cl.get_platforms()[0]
```

从输出结果可以看出，平台信息对应于 NVIDIA 的 CUDA 平台。接着，使用平台的 get_device() 方法选择设备。

```
device = platform.get_devices()[0]
```

在接下来的代码中，定义上下文和队列。PyOpenCL 提供了上下文（选中的设备）和队列（选中的上下文）方法：

```
context = cl.Context([device])
queue = cl.CommandQueue(context)
```

为了在设备上进行计算，输入向量必须转移到设备内存中。因此，必须在设备内存中创建两个输入缓冲区（input buffer）：

```
mf = cl.mem_flags
a_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=vector_a)
b_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=vector_b)
```

另外，还要准备好最终输出向量的缓冲区：

```
res_g = cl.Buffer(context, mf.WRITE_ONLY, vector_a.nbytes)
```

最后，在 program 中定义该脚本的核心，即内核代码：

```
program = cl.Program(context," ""
__kernel void vectorSum(__global const int *a_g, __global const int
*b_g, __global int *res_g) {
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
" "").build()
```

内核名称为 vectorSum，参数列表定义了输入参数的数据类型（整数向量）和输出数据类

型（整数向量）。

在内核函数的主体中，两个向量之和的定义如下所示。

- **初始化向量索引：**`int gid = get_global_id(0)`
- **向量的组件相加：**`res_g[gid] = a_g[gid] + b_g[gid]`

在 OpenCL 和 PyOpenCL 中，缓冲区依附于上下文，只有在设备上使用该缓冲区时才会转移到设备中。最后，我们在设备上执行 `vectorSum`：

```
program.vectorSum(queue, vector_a.shape, None, a_g, b_g, res_g)
```

为了可视化地查看结果，构建一个空向量：

```
res_np = np.empty_like(vector_a)
```

然后，将结果复制到这个向量中：

```
cl.enqueue_copy(queue, res_np, res_g)
```

最后，打印结果：

```
print ("VECTOR LENGTH = %s" %vector_dimension)
print ("INPUT VECTOR A")
print vector_a
print ("INPUT VECTOR B")
print vector_b
print ("OUTPUT VECTOR RESULT A + B")
print res_np
```

我们使用 `assert` 语句来检查结果。这将检测结果是否正确，如果条件为假，将引发错误：

```
assert(la.norm(res_np - (vector_a + vector_b))) < 1e-5
```

使用PyOpenCL对逐元素表达式求值

与 PyCUDA 类似，PyOpenCL 在 `pyopencl.elementwise` 类中提供了在单次计算（single computational pass）中求复杂表达式值的功能。实现该功能的方法是：

```
ElementwiseKernel(context, argument, operation, name,",",",","
                          optional_parameters)
```

- `context`：指的是执行逐元素操作的设备或设备组。
- `argument`：这是由计算中所需参数组成的类 C 参数列表。
- `operation`：这是一个字符串，代表将使用参数列表执行的操作。
- `name`：指的是与 `ElementwiseKernel` 相关联的内核名称。

- optional_parameters：这些参数对于该实例并不重要。

具体实现

在本例中，我们再次来完成将两个整数向量相加的任务，向量均包含 100 个元素。当然，最终的实现代码并不一样，因为我们用的是 ElementwiseKernel 类，如下所示：

```
import pyopencl as cl
import pyopencl.array as cl_array
import numpy as np

context = cl.create_some_context()
queue = cl.CommandQueue(context)

vector_dimension = 100
vector_a = cl_array.to_device(
    queue,  np.random.randint(vector_dimension, size=vector_dimension))
vector_b = cl_array.to_device(
    queue,  np.random.randint(vector_dimension, size=vector_dimension))
result_vector = cl_array.empty_like(vector_a)

elementwiseSum = cl.elementwise.ElementwiseKernel(
    context, "int * a, int * b, int * c", "c[i]=a[i] + b[i]", "sum")
elementwiseSum(vector_a, vector_b, result_vector)

print("PyOpenCL ELEMENTWISE SUM OF TWO VECTORS")
print("VECTOR LENGTH = %s" % vector_dimension)
print("INPUT VECTOR A")
print vector_a
print("INPUT VECTOR B")
print vector_b
print("OUTPUT VECTOR RESULT A + B")
print result_vector
```

上述代码的输出如下：

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\>python
PyOpenCLElementwise.py


Choose platform:
[0] <pyopencl.Platform 'NVIDIA CUDA' at 0x2cc6c40>
[1] <pyopencl.Platform 'Intel(R) OpenCL' at 0x3cf440>
Choice [0]:0
Set the environment variable PYOPENCL_CTX='0' to avoid being asked again.
PyOpenCL ELEMENTWISE SUM OF TWO VECTORS
```

```
VECTOR LENGTH = 100
INPUT VECTOR A
[70 95 47 53 71 52 15 10 95  5 76 40 55 87  7 18 44 72  2 42 47 86 58 87
64 79 44 94  5 54 92 21 60 67 43 92 38 49 97 14 17 35 87 94  3 17 87 24
50 43  39 71 84  7 64 60 29 74 65 82 42 35 96 80 94 57 21 56 94  8  3 94
30 64 44  34 79  5 88 80 98 88  5  2 77 57  7 93 49 42 56 19 81 36 19 24
27 18  1 40]
INPUT VECTOR B
[82 32 72  9 29 29 92  2 20 44 31 91 63 97 86 37 39 41 19 78 60 30 21 69
29  38 56 49 97 18 44 84 27 73 73 14 67 43 17 58 81 52 89 84 80 96 58 80
20 91  20 61 92 46 34 98 21 82 52 34 81 45 35 28 23 59 21 89 47 75 49 43
92 91 84  59 35 61 42 12 69 15 98 85 12 36 64 89 76 29  8 81 62  5 58 13
46 82 12 66]
OUTPUT VECTOR RESULT A + B
[152 127 119  62 100  81 107  12 115  49 107 131 118 184  93  55  83 113
  21 120 107 116  79 156  93 117 100 143 102  72 136 105  87 140 116 106
 105  92 114  72  98  87 176 178  83 113 145 104  70 134  59 132 176  53
  98 158  50 156 117 116 123  80 131 108 117 116  42 145 141  83  52 137
 122 155 128  93 114  66 130  92 167 103 103  87  89  93  71 182 125  71
  64 100 143  41  77  37  73 100  13 106]
```

实例精解

在脚本的第一行，我们导入了所有必需的模块：

```
import pyopencl as cl
import pyopencl.array as cl_array
import numpy as np
```

使用 cl.create_some_context() 方法初始化上下文。这将让用户选择使用哪个上下文执行计算：

```
Choose platform:
[0] <pyopencl.Platform 'NVIDIA CUDA' at 0x2cc6c40>
[1] <pyopencl.Platform 'Intel(R) OpenCL' at 0x3cf440>
```

然后，实例化队列，用于接收 ElementwiseKernel：

```
queue = cl.CommandQueue(context)
```

接着，创建输入向量和结果向量的实例：

```
vector_dimension = 100
vector_a = cl_array.to_device(
    queue,  np.random.randint(vector_dimension, size=vector_dimension))
vector_b = cl_array.to_device(
    queue,  np.random.randint(vector_dimension, size=vector_dimension))
```

```
result_vector = cl_array.empty_like(vector_a)
```

输入向量 vector_a 和 vector_b 都是整数向量，整数值为通过 NumPy 的 random.radint 函数获得的随机值。接着使用下面的 PyOpenCL 语句将输入向量复制到设备中：

```
cl.array_to_device(queue,array)
```

最后，创建 ElementwiseKernel 对象：

```
elementwiseSum = cl.elementwise.ElementwiseKernel(context, "int *a, int *b,
int *c", "c[i] = a[i] + b[i]", "sum")
```

在这行代码中：

- 所有参数都包含在一个字符串中，格式为 C 参数列表（均为整数）。
- 由一个 C 代码段执行操作，即将向量组件相加。
- 函数名将用于编译内核。

接下来，可使用上面定义的参数调用 elementwiseSum 函数：

```
elementwiseSum(vector_a, vector_b, result_vector)
```

本例最后会打印输入向量以及获得的结果：

```
print vector_a
print vector_b
print result_vector
```

使用 PyOpenCL 测试 GPU 应用

在本章中，我们测试了 CPU 和 GPU 之间的性能差异。在开始研究算法性能之前，要注意进行测试的执行平台。事实上，这些系统的具体特性会影响计算时间，属于需要重点考虑的因素。

我们使用如下机器进行测试。

- GPU：GeForce GT 240
- CPU：Intel Core2 Duo 2.33 GHz
- RAM：DDR2 4 GB

具体实现

在本次测试中，将计算并比较一个简单数学运算的计算时间，该运算为求两个向量之和，元素均为浮点数。为了进行比较，在两个不同的函数中实现相同的操作。

第一个函数只使用 CPU，第二个则使用 PyOpenCL 编写，并利用 GPU 进行计算。测试使用的输入为最大维度为 10 000 个元素的向量。

测试代码如下：

```
from time import time  # 导入时间工具

import pyopencl as cl
import numpy as np
import PyOpeClDeviceInfo as device_info
import numpy.linalg as la

# 输入向量
a = np.random.rand(10000).astype(np.float32)
b = np.random.rand(10000).astype(np.float32)

def test_cpu_vector_sum(a, b):
    c_cpu = np.empty_like(a)
    cpu_start_time = time()
    for i in range(10000):
        for j in range(10000):
            c_cpu[i] = a[i] + b[i]
    cpu_end_time = time()
    print("CPU Time: {0} s".format(cpu_end_time - cpu_start_time))
    return c_cpu


def test_gpu_vector_sum(a, b):
    # 定义 PyOpenCL 上下文
    platform = cl.get_platforms()[0]
    device = platform.get_devices()[0]
    context = cl.Context([device])
    queue = cl.CommandQueue(context,
                properties=cl.command_queue_properties.PROFILING_ENABLE)

# 准备数据结构
    a_buffer = cl.Buffer\
        (context,
         cl.mem_flags.READ_ONLY
         | cl.mem_flags.COPY_HOST_PTR, hostbuf=a)
    b_buffer = cl.Buffer\
        (context,
         cl.mem_flags.READ_ONLY
         | cl.mem_flags.COPY_HOST_PTR, hostbuf=b)
    c_buffer = cl.Buffer\
        (context,
         cl.mem_flags.WRITE_ONLY, b.nbytes)
    program = cl.Program(context, """
    __kernel void sum(__global const float *a,
```

```
                    __global const float *b,
                    __global float *c)
    {
        int i = get_global_id(0);
        int j;
        for(j = 0; j < 10000; j++)
        {
            c[i] = a[i] + b[i];
        }
    }""").build()
    # 开始 GPU 测试
    gpu_start_time = time()
    event = program.sum(queue, a.shape, None,
                        a_buffer, b_buffer, c_buffer)
    event.wait()
    elapsed = 1e-9*(event.profile.end - event.profile.start)
    print("GPU Kernel evaluation Time: {0} s".format(elapsed))
    c_gpu = np.empty_like(a)
    cl.enqueue_read_buffer(queue, c_buffer, c_gpu).wait()
    gpu_end_time = time()
    print("GPU Time: {0} s".format(gpu_end_time - gpu_start_time))
    return c_gpu

# 开始测试
if __name__ == "__main__":
    # 打印设备信息
    device_info.print_device_info()
    # 在 CPU 上调用测试
    cpu_result = test_cpu_vector_sum(a, b)
    # 在 GPU 上调用测试
    gpu_result = test_gpu_vector_sum(a, b)
    #
    assert (la.norm(cpu_result - gpu_result)) < 1e-5
```

测试的输出如下所示，其中打印了带执行时间的设备信息：

```
C:\Python Cook\Chapter 6 - GPU Programming with Python\Chapter 6 -
codes>python PyOpenCLTestApplication.py

============================================================
OpenCL Platforms and Devices
============================================================
Platform - Name:  NVIDIA CUDA
Platform - Vendor:  NVIDIA Corporation
Platform - Version:  OpenCL 1.1 CUDA 6.0.1
Platform - Profile:  FULL_PROFILE
```

```
    ---------------------------------------------------------
    Device - Name:  GeForce GT 240
    Device - Type:  GPU
    Device - Max Clock Speed:  1340 Mhz
    Device - Compute Units:  12
    Device - Local Memory:  16 KB
    Device - Constant Memory:  64 KB
    Device - Global Memory: 1 GB
    Device - Max Buffer/Image Size: 256 MB
    Device - Max Work Group Size: 512
=============================================================
Platform - Name:  Intel(R) OpenCL
Platform - Vendor:  Intel(R) Corporation
Platform - Version:  OpenCL 1.2
Platform - Profile:  FULL_PROFILE
    ---------------------------------------------------------
    Device - Name:  Intel(R) Core(TM)2 Duo CPU     E6550  @ 2.33GHz
    Device - Type:  CPU
    Device - Max Clock Speed:  2330 Mhz
    Device - Compute Units:  2
    Device - Local Memory:  32 KB
    Device - Constant Memory:  128 KB
    Device - Global Memory: 2 GB
    Device - Max Buffer/Image Size: 512 MB
    Device - Max Work Group Size: 8192



CPU Time: 71.9769999981 s
GPU Kernel Time: 0.075756608 s
GPU Time: 0.0809998512268 s
```

虽然测试的计算开销并不太大，还是能明显地说明 GPU 显卡的计算潜力。

实例精解

如上所述，测试包含两部分，即在 CPU 上运行的代码和在 GPU 上运行的代码。运行两种代码，并比较执行时间。

对于在 CPU 上运行的测试，实现了一个 `test_cpu_vector_sum` 函数，由对 10 000 个向量元素的两个循环组成：

```
cpu_start_time = time()
  for i in range(10000):
              for j in range(10000):
         c_cpu[i] = a[i] + b[i]
```

```
  cpu_end_time = time()
```

对第 *i* 个向量组件的求和操作将执行 10 亿次，会消耗巨大的计算资源。

CPU 总运行时间将是以下两个时间之差：

```
CPU Time = cpu_end_time - cpu_start_time
```

为了测试 GPU 版本的运行时间，我们针对 PyOpenCL 实现了一个应用的常规定义：

- 确定设备和上下文的定义。
- 设置执行队列。
- 在设备上创建用于执行计算的内存区域（三块缓冲区，分别定义为 a_buffer、b_buffer、c_buffer）。
- 构建内核。
- 执行内核调用：

```
  gpu_start_time = time()
                event = program.sum(queue, a.shape, None,
                           a_buffer, b_buffer, c_buffer)

                cl.enqueue_read_buffer(queue, c_buffer, c_gpu).wait()
    gpu_end_time = time()
```

这里，GPU Time = gpu_end_time - gpu_start_time。

最后，我们在主程序中调用测试函数，以及此前定义的 print_device_info()：

```
if __name__ == "__main__":
    device_info.print_device_info()
    cpu_result = test_cpu_vector_sum(a, b)
    gpu_result = test_gpu_vector_sum(a, b)
    assert (la.norm(cpu_result - gpu_result)) < 1e-5
```

接着使用了 assert 语句检查结果，可验证结果是否正确，如果条件为假则引发错误。